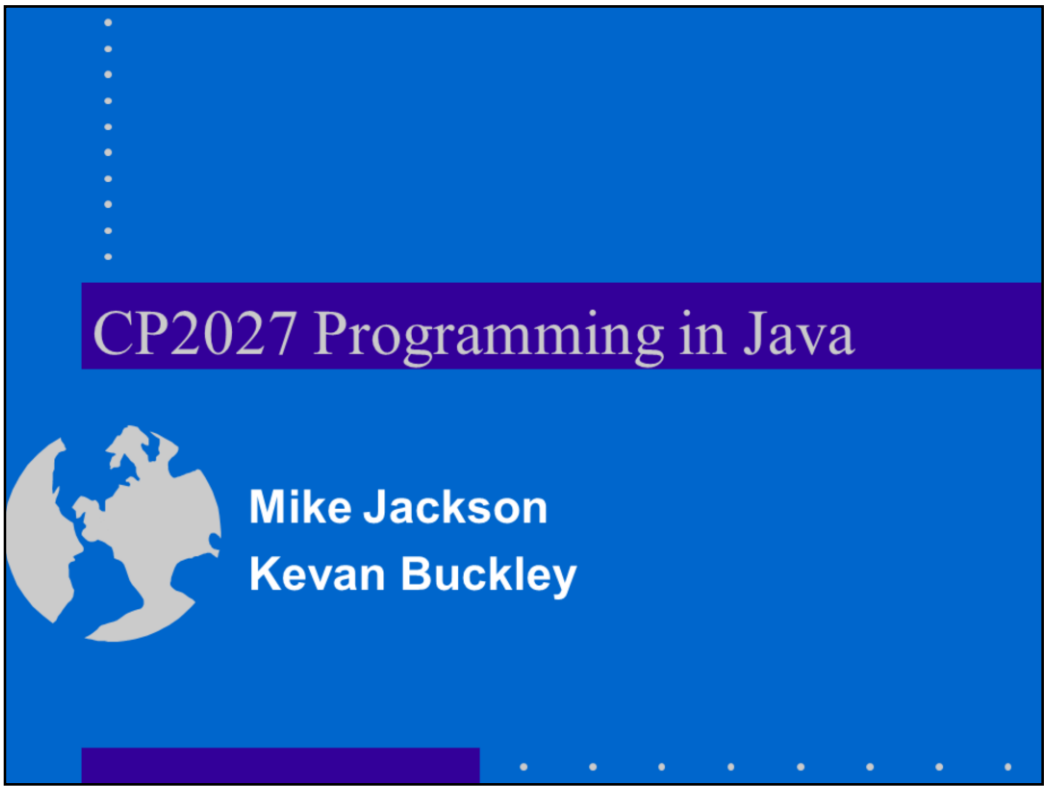


Written by Mike Jackson and Naomi Wrighton.

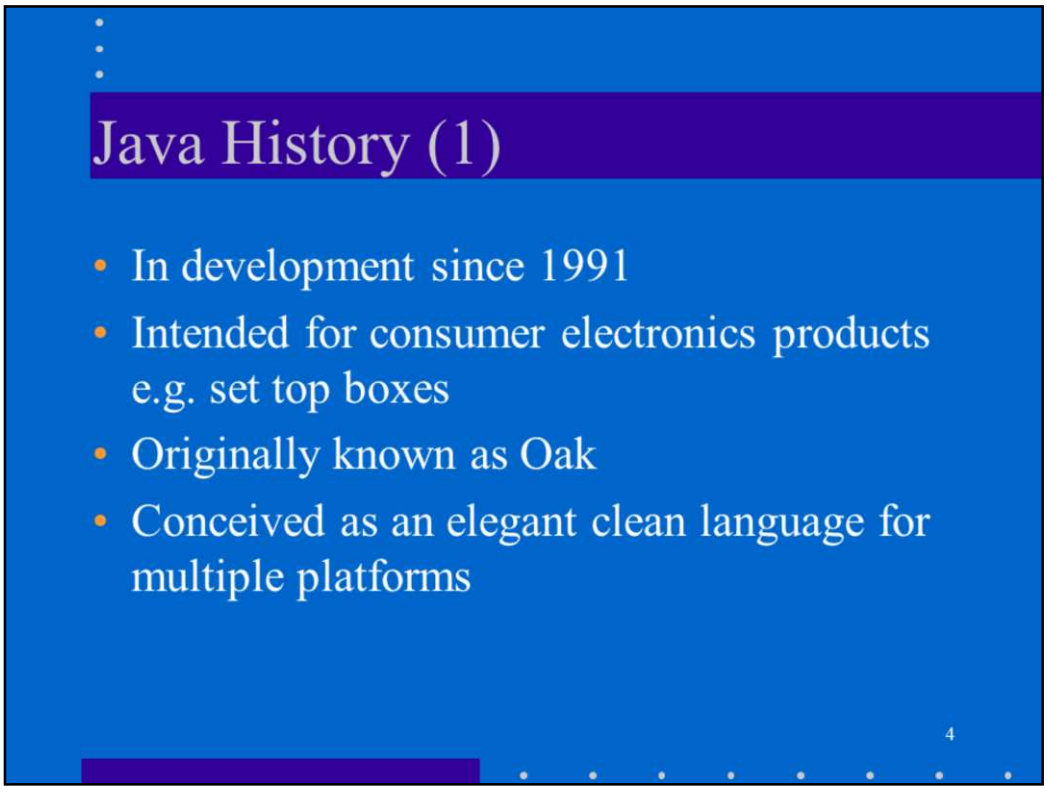
Updated by Kevan Buckley

Lecture 1

Basic Concepts

Contents

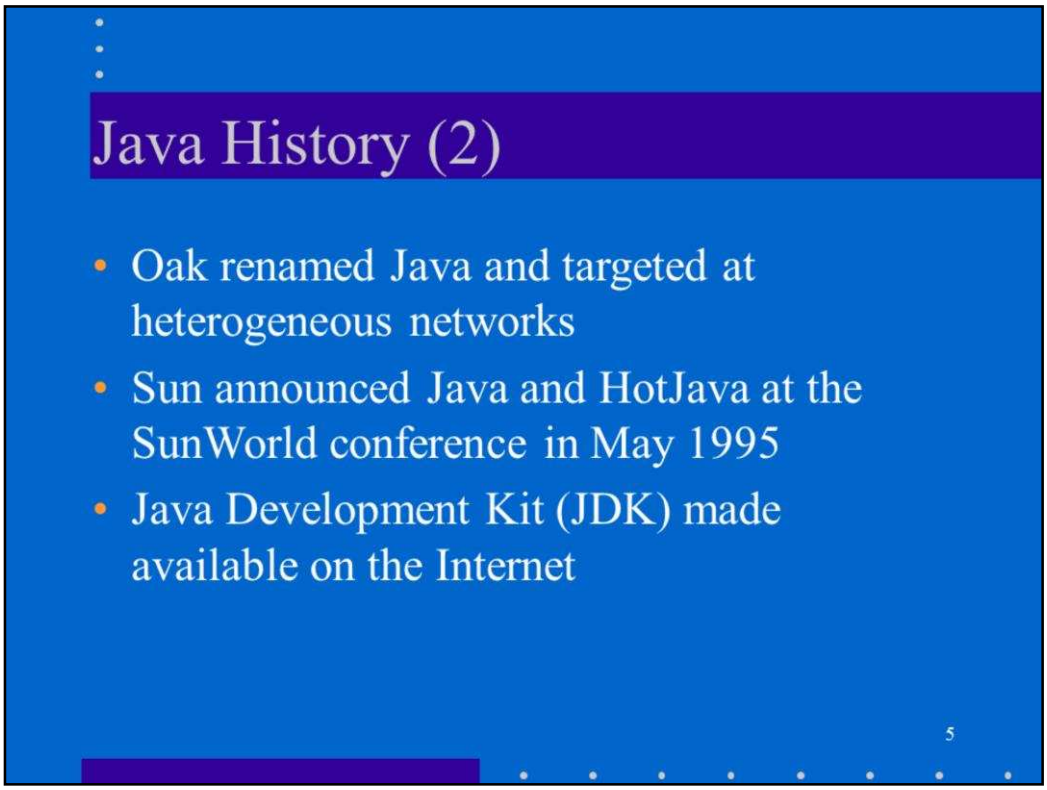
- **Introduction to Java**
  - applications and motivation
- **Java syntax**
  - literals, numbers, characters and booleans
  - declarations, scope and blocks
  - operators
  - flow of control


Java History (1)
• In development since 1991
• Intended for consumer electronics products e.g. set top boxes
• Originally known as Oak
• Conceived as an elegant clean language for multiple platforms
4

Java was developed by Sun Microsystems. Sun was founded in 1982 as a company whose primary market was Unix workstations. Since the founding of the company Sun has emphasised the value of networking. Sun is a major player in the workstation market and is one of the largest suppliers of non-PC computers (i.e non Intel and Microsoft).

The originator of Java (James Gosling) named the language Oak after the oak tree growing outside his office window. Oak was designed to be an elegant, object-oriented language which would run on multiple platforms (a variety of hardware). Oak had features which made it suitable for complex remote controls and other hardware where it would not be possible to load software from floppy disks etc.

The key characteristic of Oak was its hardware independence. Oak could run on any hardware which could support its virtual machine. Once code for virtual machine support had been ported to the platform then any Oak application would run.


Java History (2)
• Oak renamed Java and targeted at heterogeneous networks
• Sun announced Java and HotJava at the SunWorld conference in May 1995
• Java Development Kit (JDK) made available on the Internet
5

It turned out that there already was a a language called Oak, so Oak was renamed Java. The name was chosen to make people think of something that was modern and classy.

The anticipated market for Java did not develop at the rate Sun had hoped but the Internet and the World Wide Web came into prominence instead. Java was re-targetted at this market. Since the Internet is made up of machines which have many different architectures and Java is a language which will run on top of many different architectures the two seemed made for each other.

Sun formally announced Java in May 1995. They made the Java Development Kit (JDK) available free on the Internet (from http://java.sun.com). This is a very rudimentary Java environment which just consists of a command line compiler and a run time environment but it represents the current standard Java implementation and is frequently updated. Sun also announced a Java enabled browser HotJava.

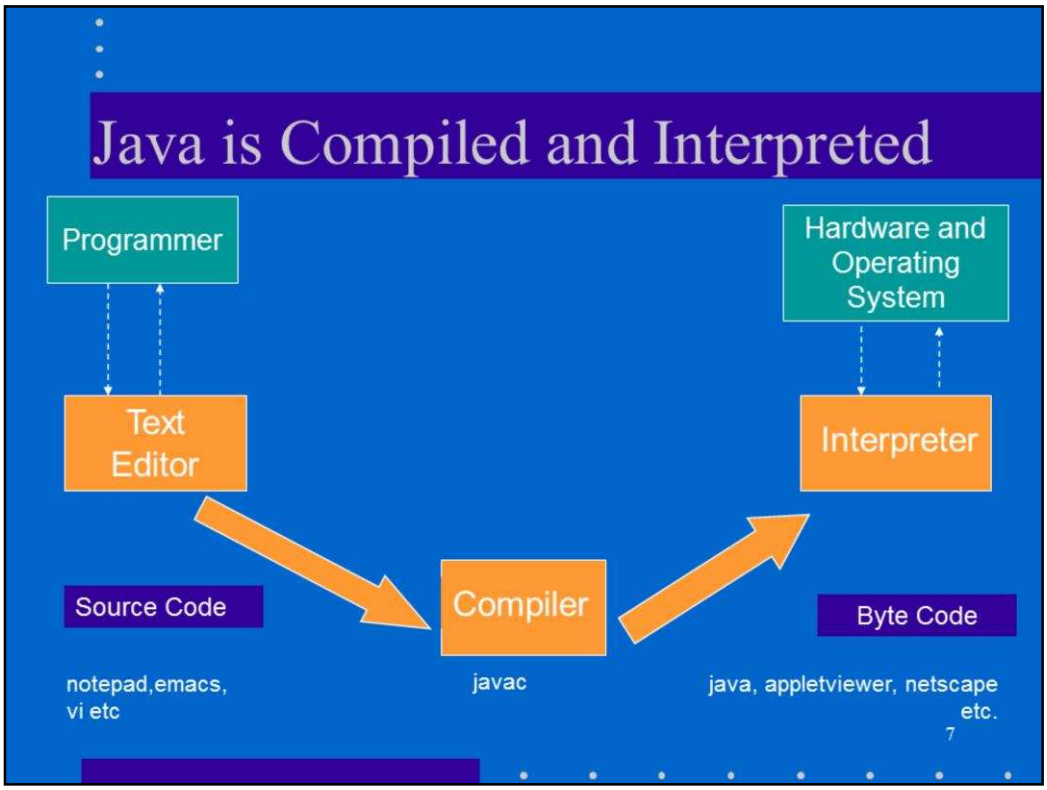
The initial excitement of Java came about because it made it possible to add dynamic behaviour to Web pages. The main browsers (Netscape and Internet Explorer) quickly began to include Java Virtual Machines which allow the transportation and execution of Java code across the Internet.

Whilst the integration of Java applets into Web pages was the initial major selling point for Java, it is probably not where the future of Java lies. Java's major strength is it's ability to run on a variety of hardware.

Java was designed so that programmers would find it simple to use. To this end a number of the more complicated features of C++ such as pointers, multiple inheritance, and operator overloading were omitted from Java. Java is a language of the 1990's and as such was designed to be object-oriented because its designers could not conceive how a good language could avoid being object-oriented! The syntax of Java was designed to be as close to C++ as possible so as to provide a ready market of Java programmers.

It was recognised that a language for the Internet could present immense security worries. Users would not want to download routines to their machines if these routines had the potential to wreck their working environments. The creators of Java built into the language mechanisms which prevent remotely loaded routines from taking control of the machine they run on.

The design of Java makes no assumptions about the machine it runs on. In fact every Java program runs on a virtual machine which is created in software. Any given hardware architecture which runs a Java program must first load an appropriate Java Virtual Machine (JVM). The JVM maps the instruction set that Java expects to the instruction set of the actual platform in use. Consequently Java is an *interpreted* rather than a *compiled* language. This means that it potentially executes slower than, say, C++. In a modern GUI environment, however, where much of the execution time of any program is taken up by operating system actions the performance loss due to interpretation is not as great as one might expect.



There are two conventional approaches to executing programs: compilation and interpretation. Java uses both.

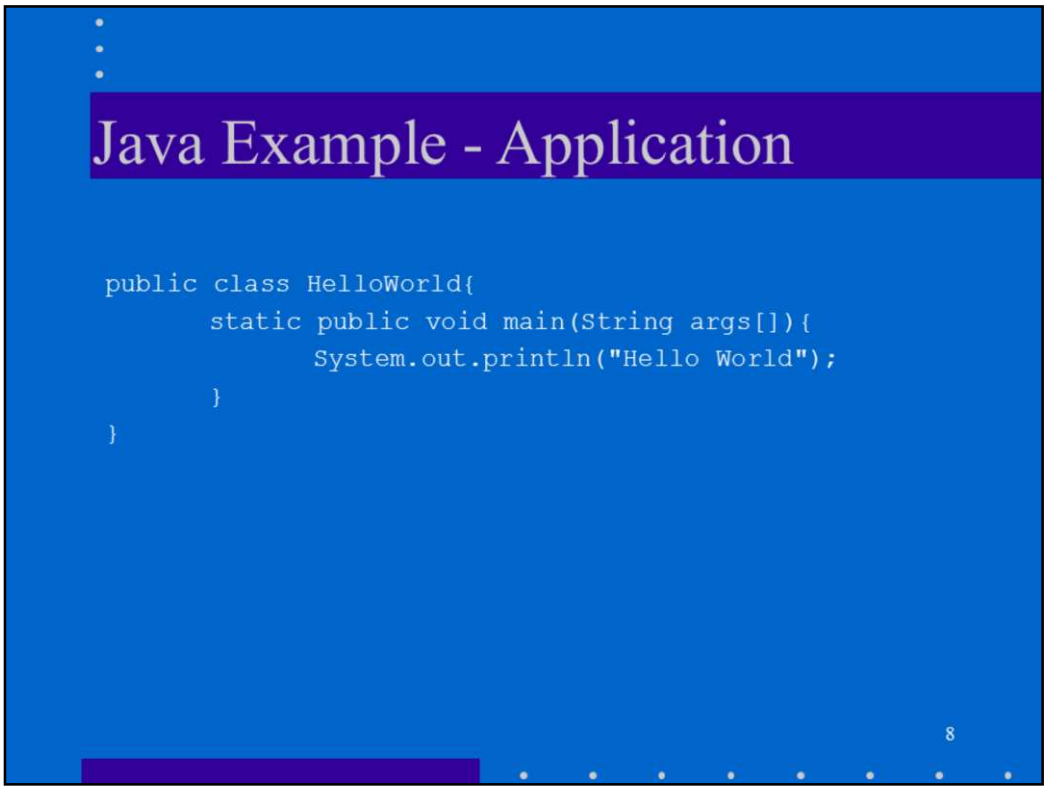
In compilation a compiler converts the program's source to into machine code that can be directly executed on the computer. This is the approach taken by C++.

In interpretation and interpreter parses and executes the source code without generating and machine code. This is the approach taken by Visual Basic.

Compiled programs obviously are quicker as they use instructions that are readily understood by the computer. Interpreted programs are run inside the interpreting program so are much slower. However, the interpreter acts like a buffer between the program and the computer and shields the computer from erroneous instructions that could affect the operation of the computer. Errors in compiled programs can jeopardise the operation of the computer as there is little to catch errors. i.e. it is the responsibility of the operating system to trap errors.

In order to allow Java programs to be platform independent (and to enforce security policies) they cannot be compiled into native machine code, thus an interpretter is required. However, unlike the BASIC language Java programs are compiled into a form that is more easily interpretted. This compiled form is called *byte code* and is executed in an interpretter called a *Java Virtual Machine (JVM)*. Byte code can be executed on any platform (architecture and operating system) that has a JVM. Almost all modern platforms have JVM's available. You can write your programs on your Windows 95

PC at home and bring them to the University and they will run on the University's UNIX servers!



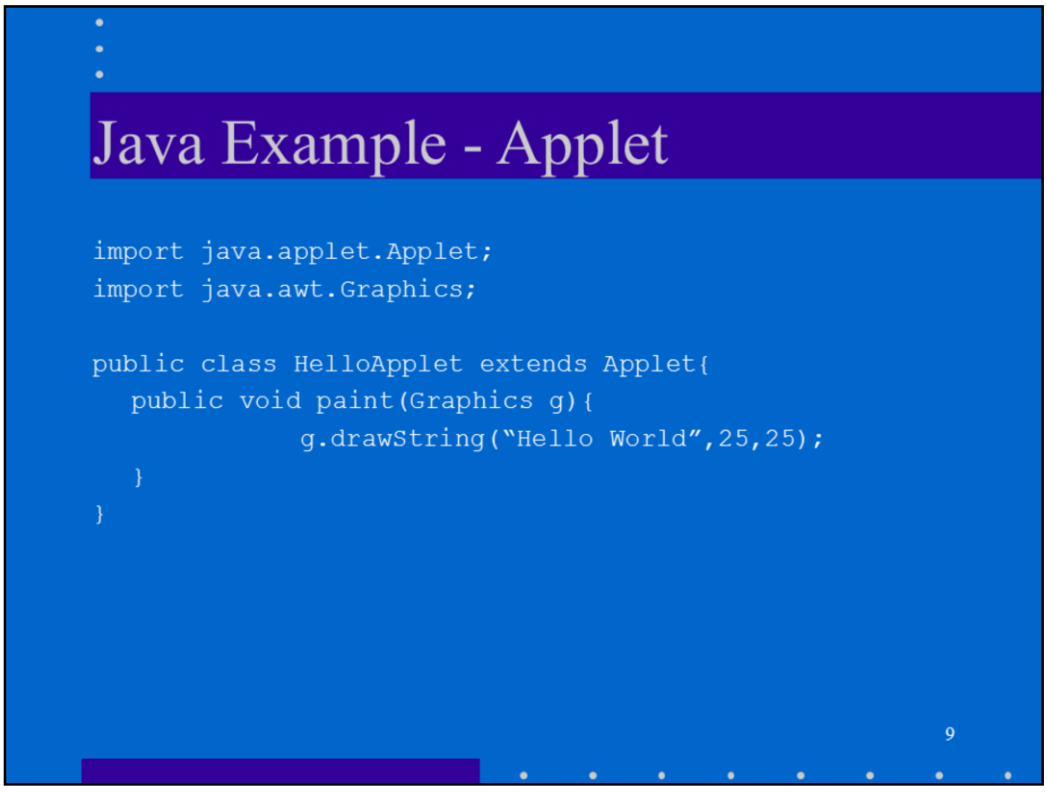
Unlike C++, Java is uncompromisingly object-oriented. Every Java application consists of a set of classes. The program above prints out the string "Hello World". All of this is encapsulated within a class. Note that a Java *application*, like a C++ program, must have a routine called main. This is the routine which is executed first when the application is run. The main routine accepts a string containing arguments, again like C++.

This application has one executable statement. The statement invokes the println method of the object represented by System.out with the argument "Hello World". The result of running this program is that

Hello World

is printed out.

Every class may have a main method (although not every class will be a fully blown application). Often main is used to contain testing code for the class.

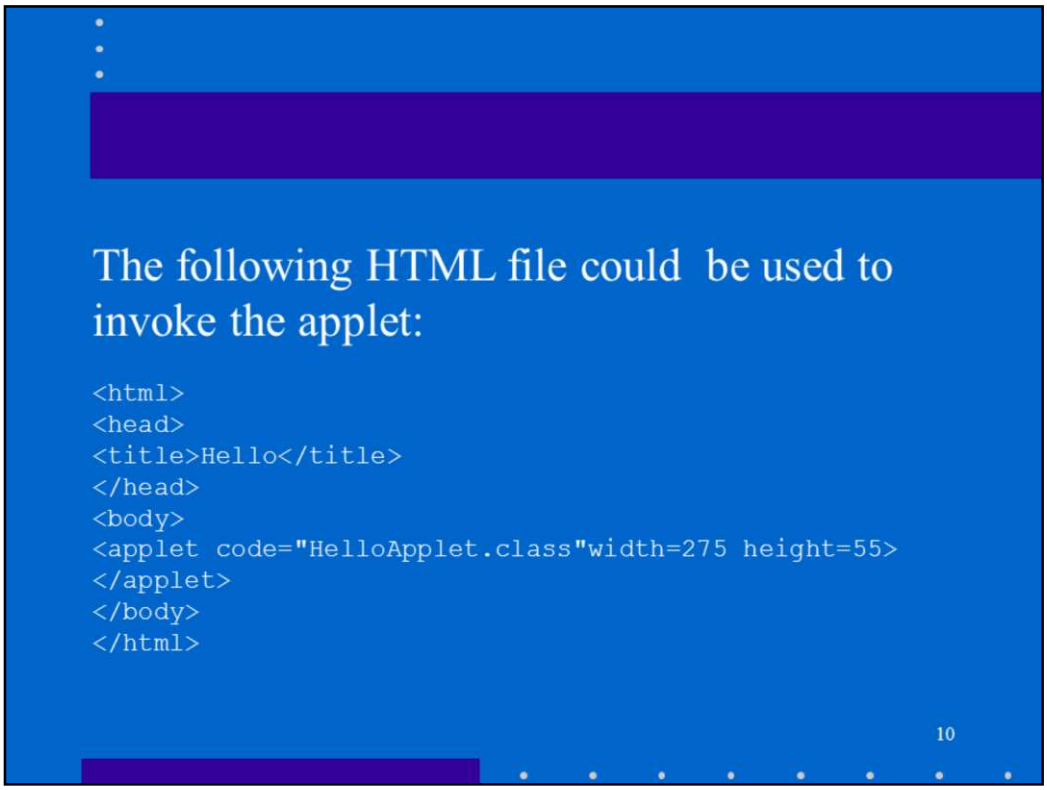


This code is for an *applet* rather than an *application*. The semi-compiled version of this code could be transmitted over the Internet and executed by a browser.

The code makes use of two Java classes `Applet` and `Graphics`. The definition of these classes must be imported. The two `import` statements act in a way which is similar to (but better than) the C and C++ include file mechanism.

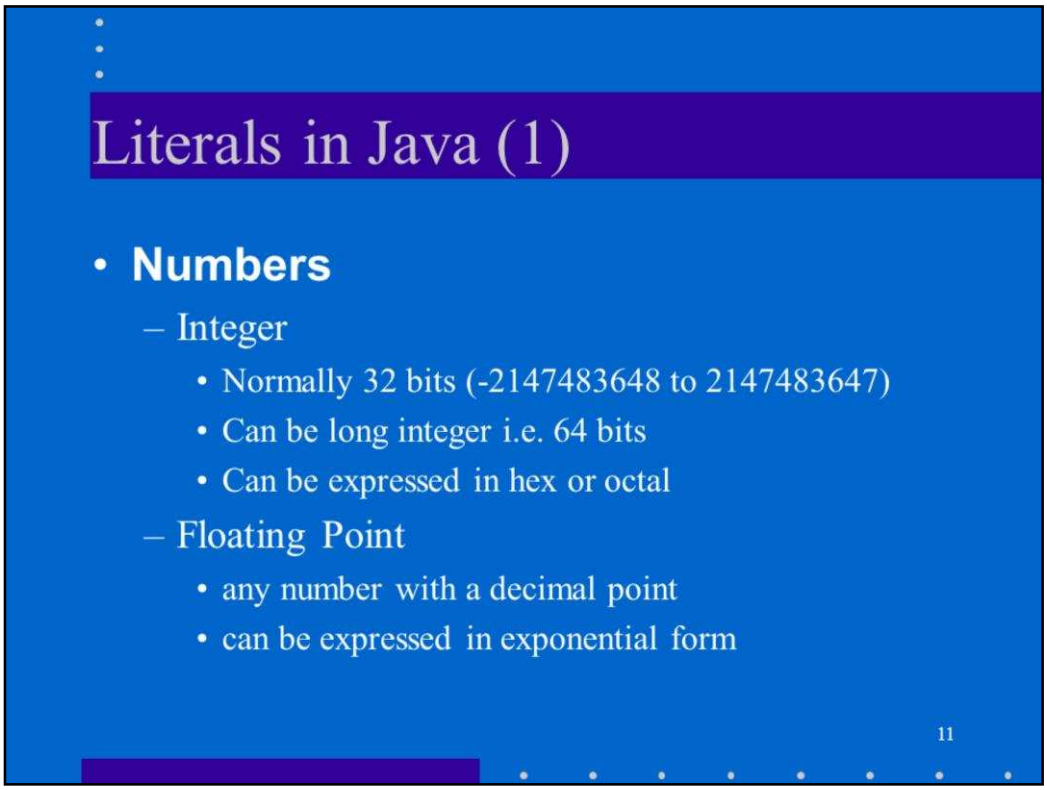
The `HelloApplet` class *inherits* from the `Applet` class. In other words it adds new methods. The keyword `extends` indicates the inheritance.

Applets do not have a method called main as they are not executed in the same way that applications are. The applet shown simply provides a redefinition of the `paint()` method. The argument to this is of type `Graphics`. When the applet is executed it will call the `paint()` method and pass an object to it which represents the graphics environment it has created. The `paint()` method makes use of the `drawString()` method provided by a graphics environment to write a string onto the the graphics area at position 25,25.



This file contains very basic HTML(HyperText Markup Language). The addition to HTML which invokes a Java program is the `applet` tag. This tag specifies a Java class file. Although Java is interpreted when it is run, its still has to be compiled. The Java compiler translates Java source code into *bytecodes*. The bytecodes are stored in a file with extension .class These bytecodes can be executed on a Java Virtual Machine (which is a bytecode interpreter). Browsers which can execute Java applets contain a version of the Java Virtual Machine.

Applet bytecode files are located on the same server which delivers the HTML file i.e. they can be remote. In this example the bytecode file, HelloApplet.class, has been stored in the same directory as the HTML file. The width and height parameters of the applet tag specify how big a space the visual representation of the applet will occupy.


Literals in Java (1)
• Numbers
– Integer
• Normally 32 bits (-2147483648 to 2147483647)
• Can be long integer i.e. 64 bits
• Can be expressed in hex or octal
– Floating Point
• any number with a decimal point
• can be expressed in exponential form
11

As we shall see later Java has a number of built-in types. These are `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`.

Each of these can be expressed by a literal.

Numbers without decimal points are assumed to be integers. For example:

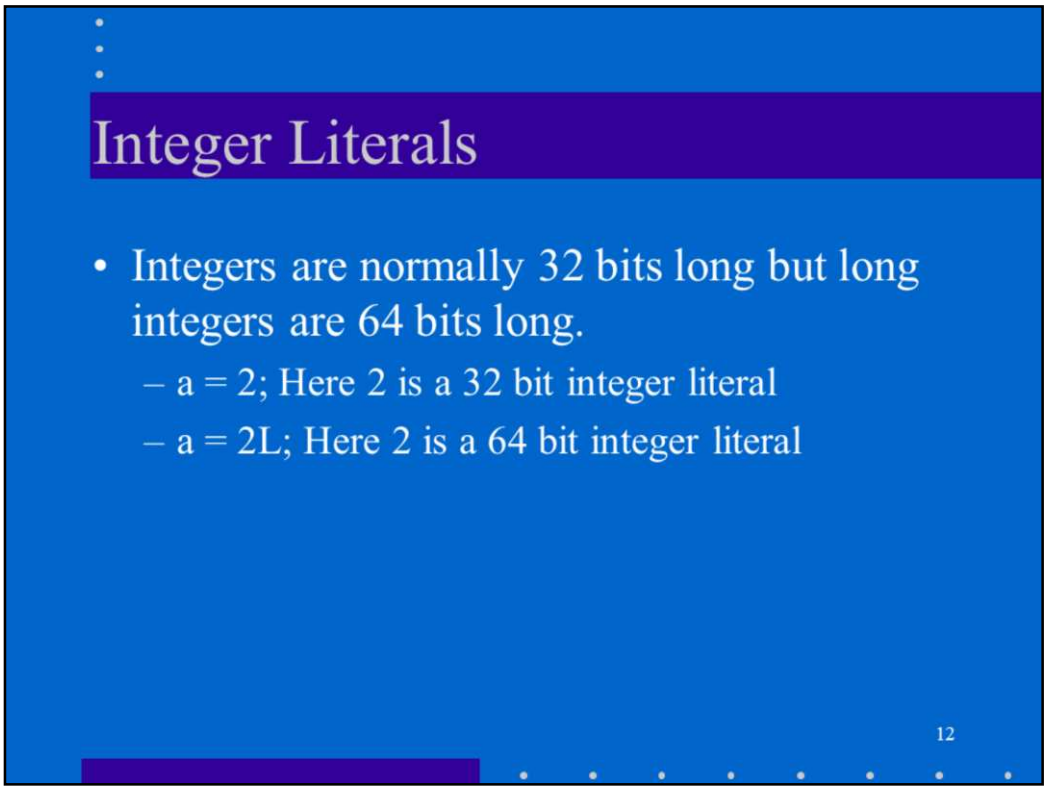
```
a = 3;
```

sets the value of a to an `int` of value 3

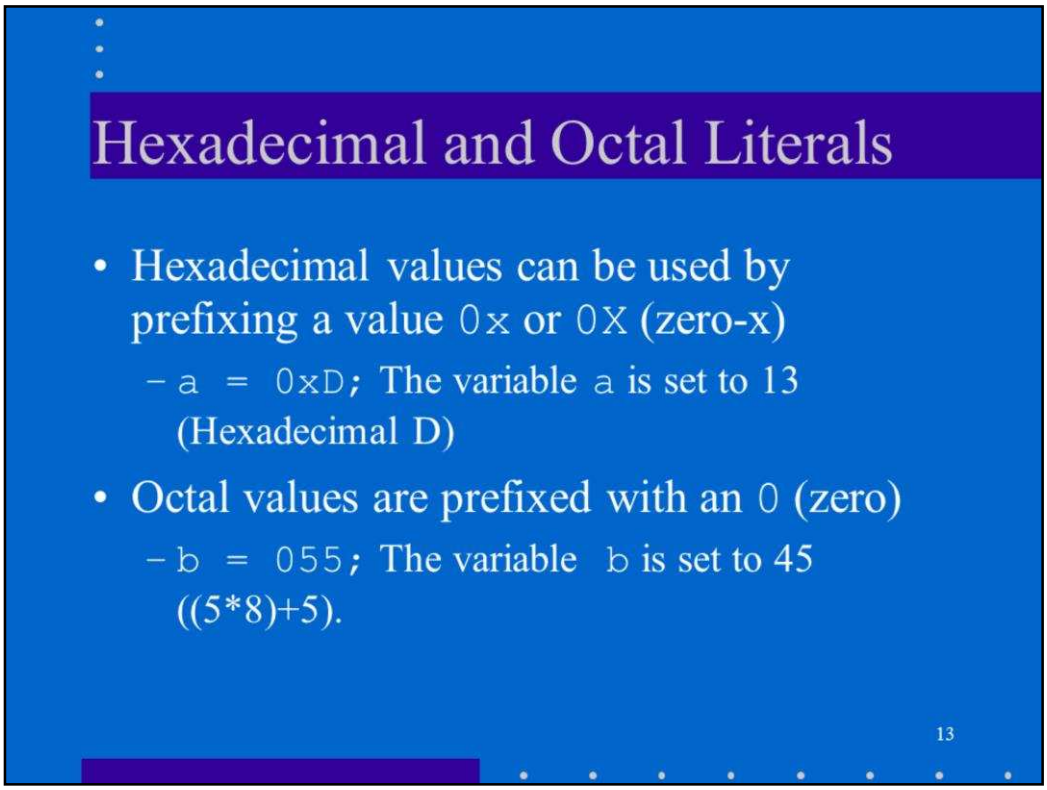
Any number with a decimal point in it is a floating point literal. For example:

```
a = 2.5;
```

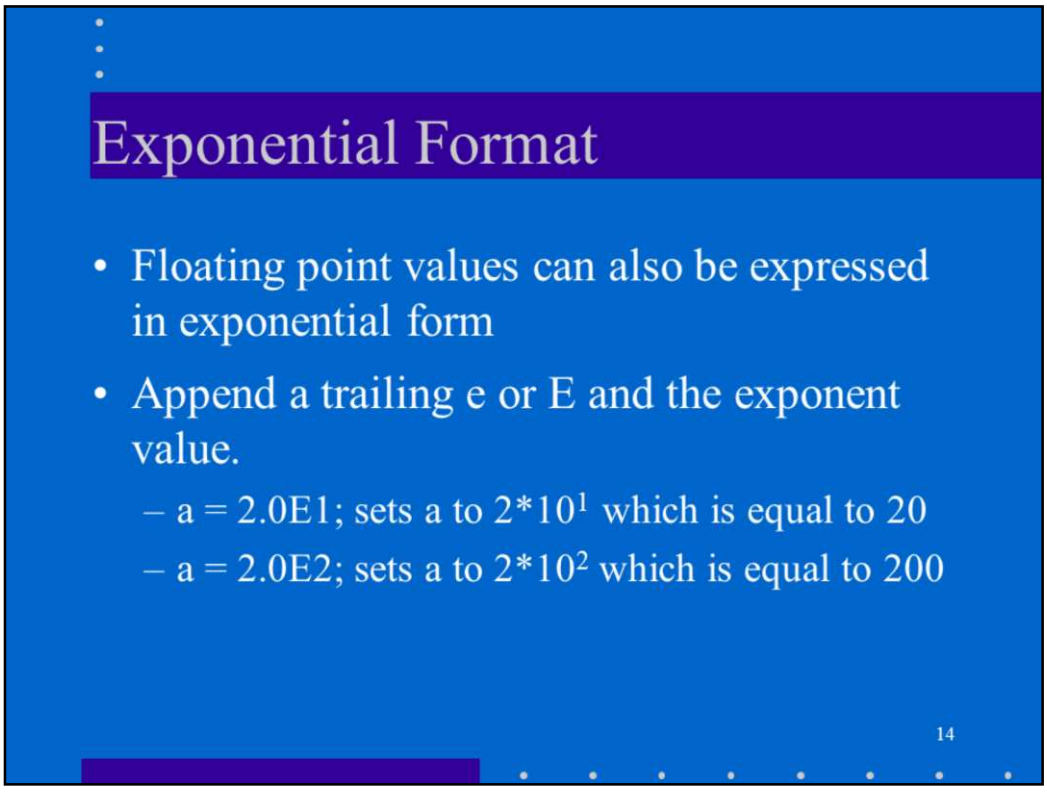
sets the value of a to be a `double` of value 2.5.


Integer Literals
• Integers are normally 32 bits long but long integers are 64 bits long.
– a = 2; Here 2 is a 32 bit integer literal
– a = 2L; Here 2 is a 64 bit integer literal
12

The normal size integer require 32 bits of storage but 64 bit integers can be used for very large negative or positive whole numbers.

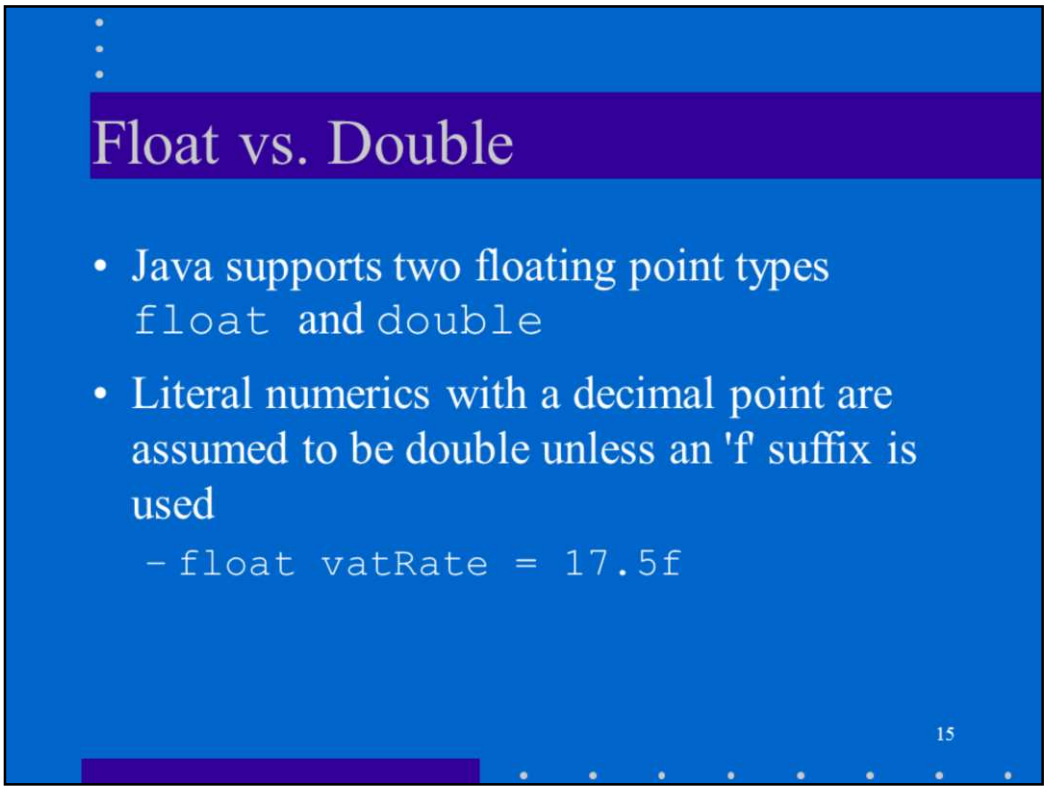

Hexadecimal and Octal Literals
• Hexadecimal values can be used by prefixing a value 0x or 0X (zero-x)
– a = 0xD; The variable a is set to 13 (Hexadecimal D)
• Octal values are prefixed with an 0 (zero)
– b = 055; The variable b is set to 45 ((5*8)+5).
13

The mechanism for using constants in Hexadecimal or Octal (a thing systems programmers like to do) is similar to the one used in C and C++.


Exponential Format
• Floating point values can also be expressed in exponential form
• Append a trailing e or E and the exponent value.
– a = 2.0E1; sets a to 2*10^1 which is equal to 20
– a = 2.0E2; sets a to 2*10^2 which is equal to 200
14

Large floating point numbers require a lot of digits if written out in full. Java supports a widely accepted scientific shorthand mechanism for expressing floating point numbers.

In this format the number nEx means $n*10^x$. Examples are on the slide above. Literals of type `float` still need to have an `f` suffix when expressed in this form (see next slide).

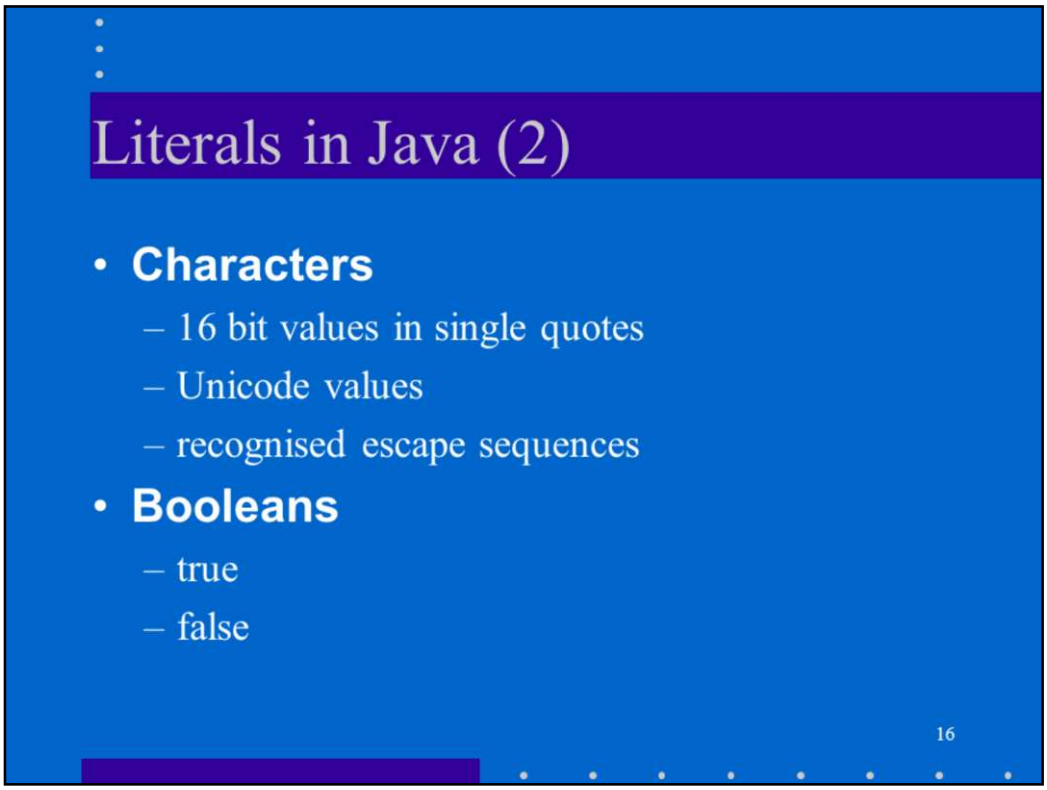

Float vs. Double
• Java supports two floating point types float and double
• Literal numerics with a decimal point are assumed to be double unless an 'f' suffix is used
– float vatRate = 17.5f
15

The default type for floating point numbers is double.

A declaration of the format:

```
float vatRate = 17.5;
```

will cause the Java compiler to output an error message which says something about the necessity of undertaking an explicit cast!



Most languages represent characters in 8 bit ASCII. This is fine for Western character sets. Java was designed to be truly international and able to cope with Arabic, Japanese, Chinese and other characters. It must therefore support a much larger number of characters. Characters in Java are encoded as 16 bit Unicode values. Details of Unicode can be found at http://unicode.org.

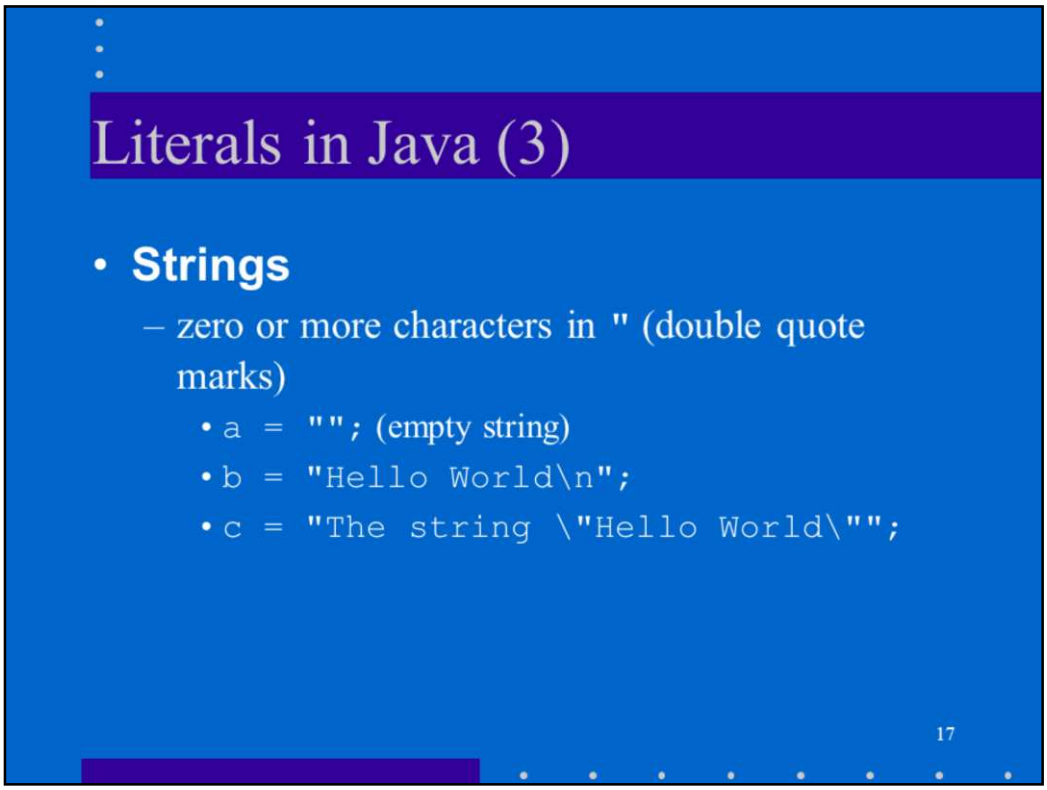
As in C and C++ character literals are represented by single characters within single quotes.

```
a = 'c';
```

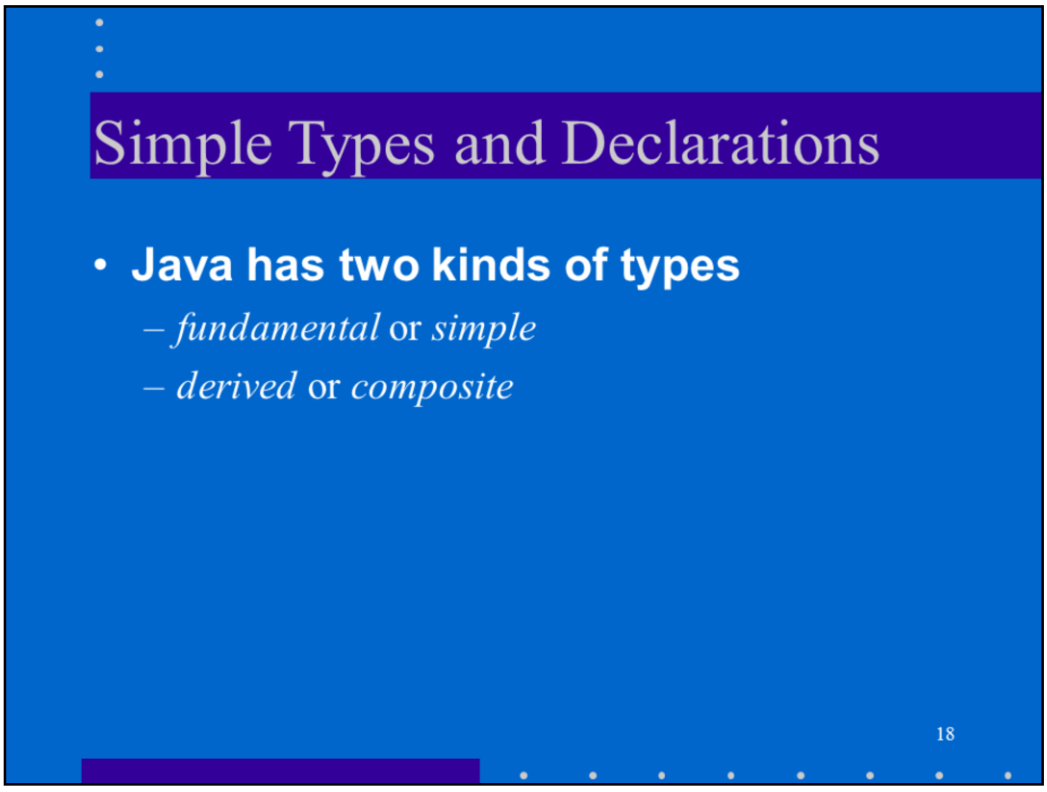
There are a number of familiar escape sequences for special characters:

`\b`	backspace	`\f`	formfeed
`\n`	newline	`\r`	carriage return
`\t`	horizontal tab	`\v`	vertical tab
`\\`	backslash	`\?`	question mark
`\'`	single quote	`\"`	double quote
`\000`	octal number	`\x`hh	hex number

Languages such as C and older versions of C++ do not support variables which can only be true or false. Java supports these directly as booleans and they may only be assigned the literals `true` or `false`.


Literals in Java (3)
Strings
– zero or more characters in " (double quote marks)
• a = ""; (empty string)
• b = "Hello World\n";
• c = "The string \"Hello World\"";
17

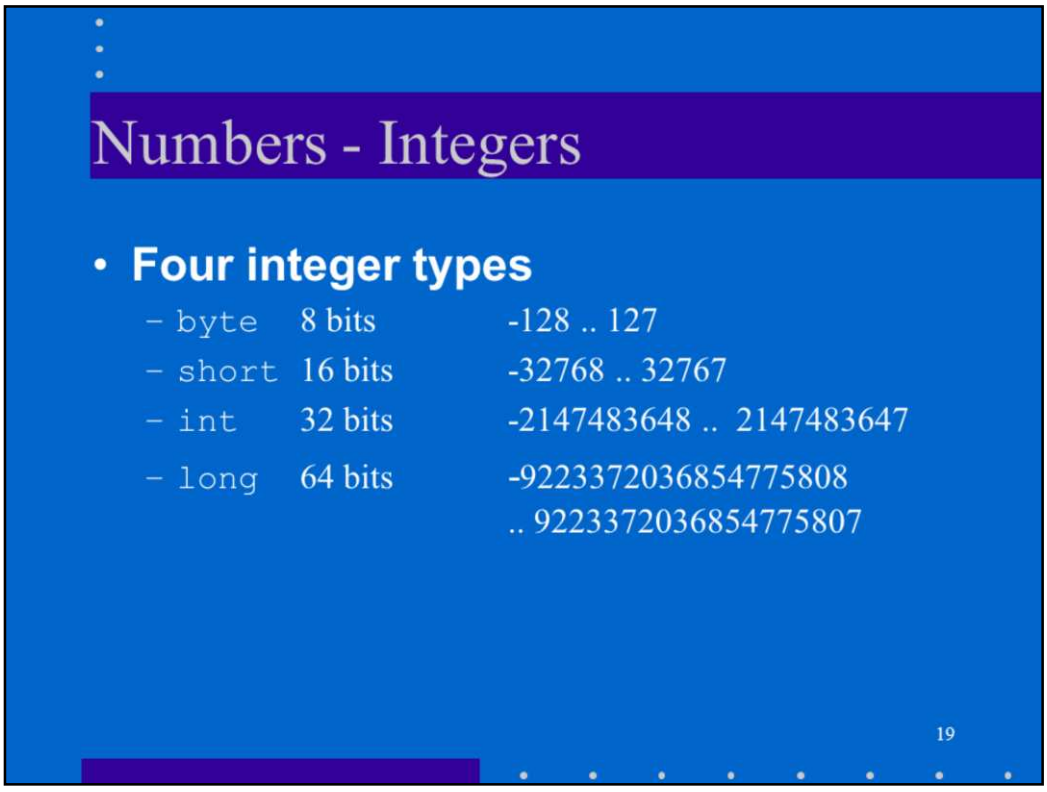
Strings in Java are much more powerful than those in C or the original versions of C++, however initially they look much the same. A string (as opposed to a character) is enclosed in double quotes. The empty string contains no characters at all. Escape characters may be used within string literals.



Pure object-oriented languages such as Smalltalk have no built-in types. In these languages everything is an object. Languages like C++ have built-in types which are not objects. Java compromises between these two approaches, it has built-in types but each of these types is mirrored by a class which acts as a factory for objects with the same properties as the built-in types.

The built-in types are said to be fundamental or simple types. This course will initially look at the simple types only but will quite quickly look at the way in which composite types can be constructed.

The next few slides deal with simple types.



A range of integer variable types are available the difference between them is the number of bits they need and the size of the integers they can represent. Languages such as C and C++ have similar choices for representing integers. The main difference between these languages and Java is that all implementations of Java will adhere to the bit representations shown. In C++ and C the compiler implementers can decide how many bits will represent each type. This inhibits language portability. The creators of Java attempted to ensure that programs written in the language would be portable by standardising how it will be implemented.

Numbers - Floating Point

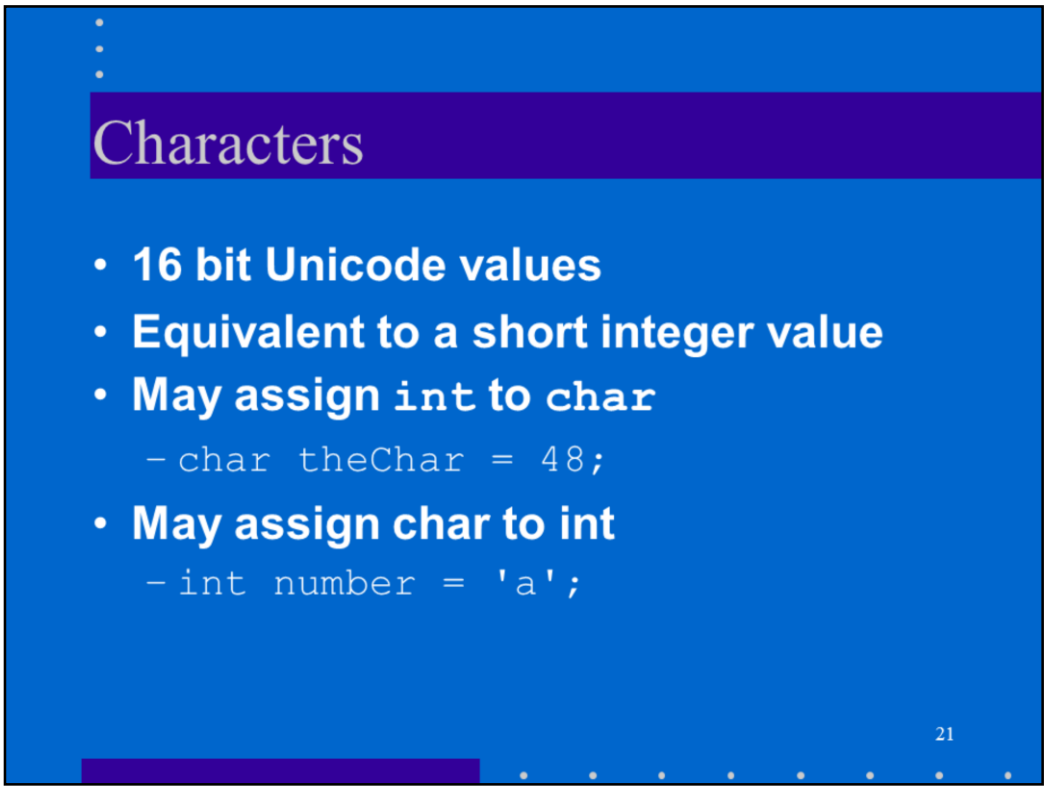
- **Two types float and double**

	Float	Double
Size	32 bits	64 bits
"Largest Value"	±3.4E+38	±1.79E308
"Smallest Value"	±1.4E-45	±4.9E-324
Precision	6-7 sig. figs	14-15 sig. figs.

20

There are two types of floating point numbers available in Java; `float` and `double`. The table on the slide compares the two. In a comparison of number of bits, `double` values occupy twice as many as `float` values. On the other hand, `double` variables can hold a greater range of numbers than `float` variables and can be more accurate.

As with integer variables, the implementation of floating point numbers is standardised on all hardware platforms.


Characters
• 16 bit Unicode values
• Equivalent to a short integer value
• May assign int to char
– char theChar = 48;
• May assign char to int
– int number = 'a';
21

Characters in Java are 16 bit values. This is a point at which Java varies from most other languages. In many other languages characters are encoded as 8 bit ASCII. In Java characters are encoded in Unicode which permits a much larger character set to be represented. Fortunately if you ignore the top 8 bits of a Unicode character you revert to 8 bit ASCII so the the two standards are compatible.

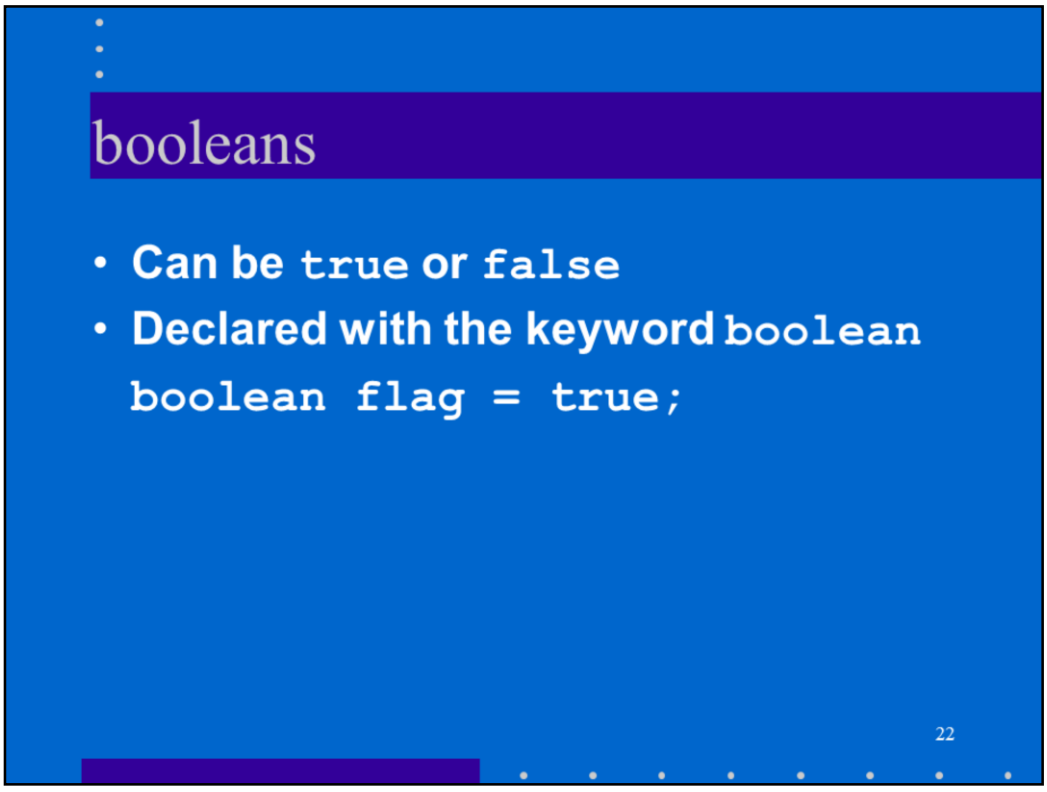
As in C and C++ Java allows characters and integers to be interchanged freely.

This is an example of an *implicit cast* i.e the language allows mixed type expressions and handles the type conversion automatically.

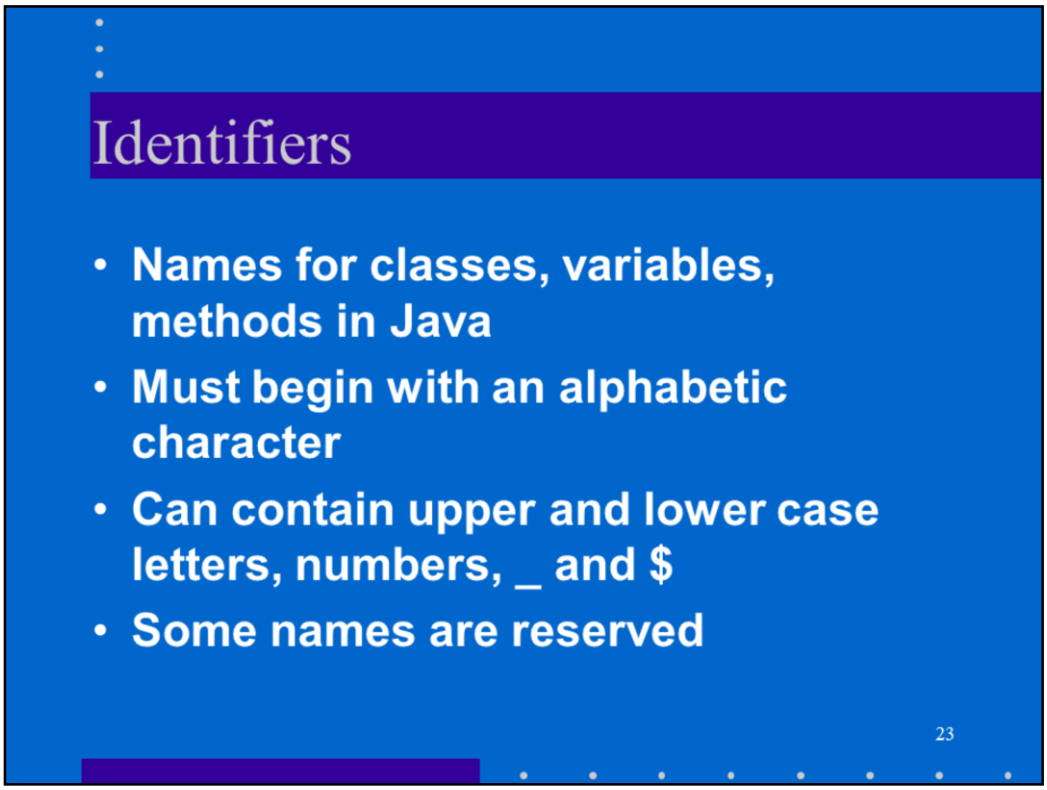
It is safer to use *explicit casts* as this makes clear to anyone else reading the code what is going on. Java uses the C cast notation.

e.g. `char theChar = (char) 48;`

The type name in brackets indicates that the `int` value 48 is being converted into an equivalent `char` value.



The result of any logical operation e.g. `6 > 7` will be either true or false (in this case false!). Java provides a variable type which can be used to hold the result of logical operations. This is called `boolean`, `boolean` variables can only hold the value `true` or `false`.

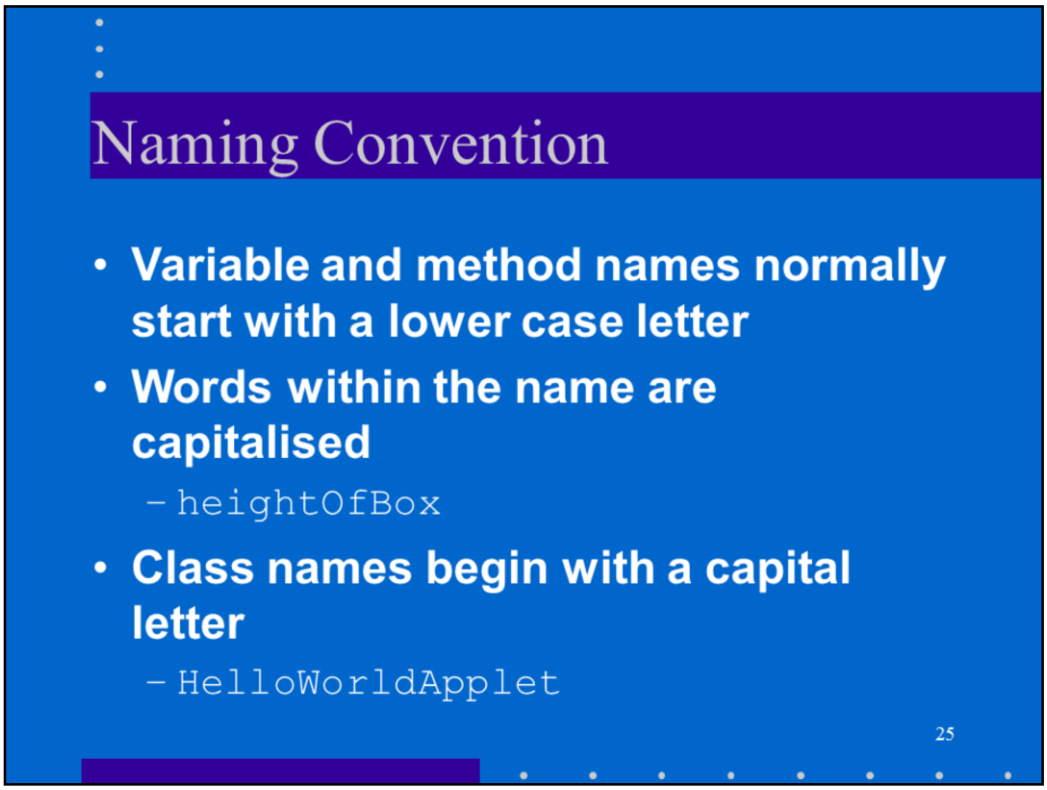

Identifiers
• Names for classes, variables, methods in Java
• Must begin with an alphabetic character
• Can contain upper and lower case letters, numbers, _ and $
• Some names are reserved
23

Like most other programming language Java allows the programmer to assign names to the various elements used in a program. Names (identifiers) in Java follow standard rules. They must begin with an alphabetic character (a-z, A-Z). They can only contain alphabetic characters, numeric characters and the underscore character (_) and the dollar ($) character. Spaces are not allowed in identifier names. Identifiers can be of any length. Upper and lower case can be used but remember that `hello` is a different variable from `Hello`, in other words case is significant.

Some words are reserved words i.e. Java uses them for something else and these cannot be used as identifiers. Examples are `int`, `class`, `for`, `if`.

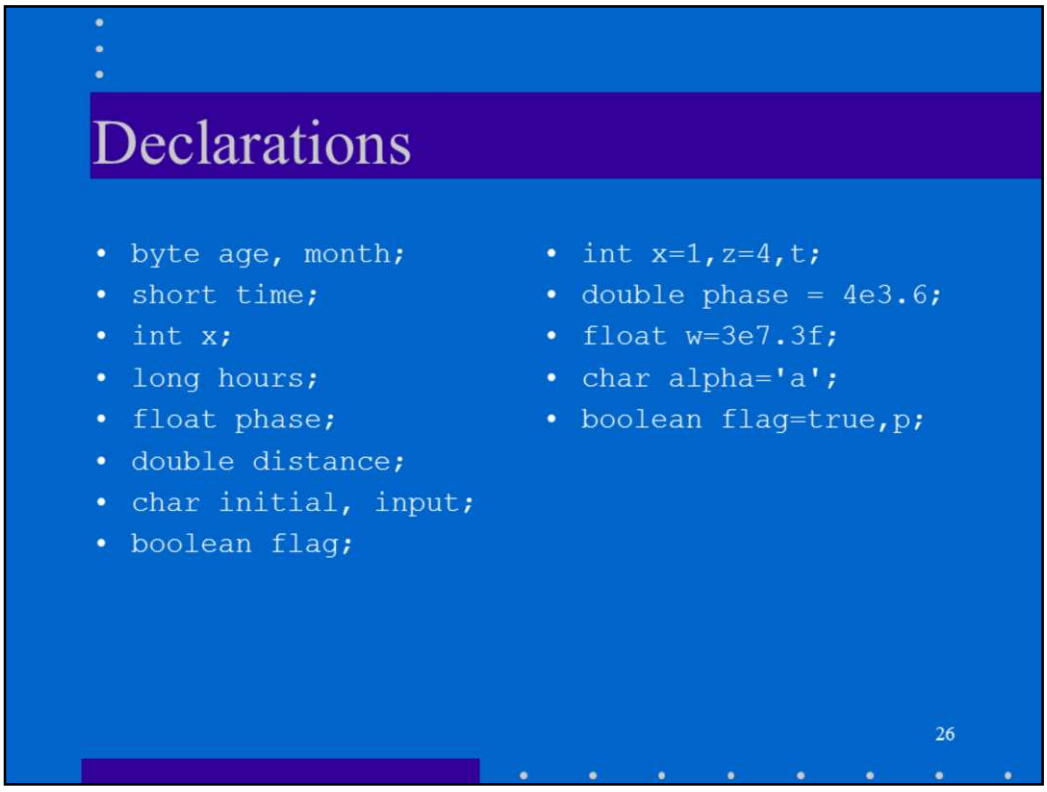
Examples

- `face` is a valid identifier
- `1face` is invalid, it begins with a number
- `face%` is invalid because % is not allowed
- `face3` is valid
- `Face` is valid but is not the same as `face`


Naming Convention
• Variable and method names normally start with a lower case letter
• Words within the name are capitalised
– heightOfBox
• Class names begin with a capital letter
– HelloWorldApplet
25

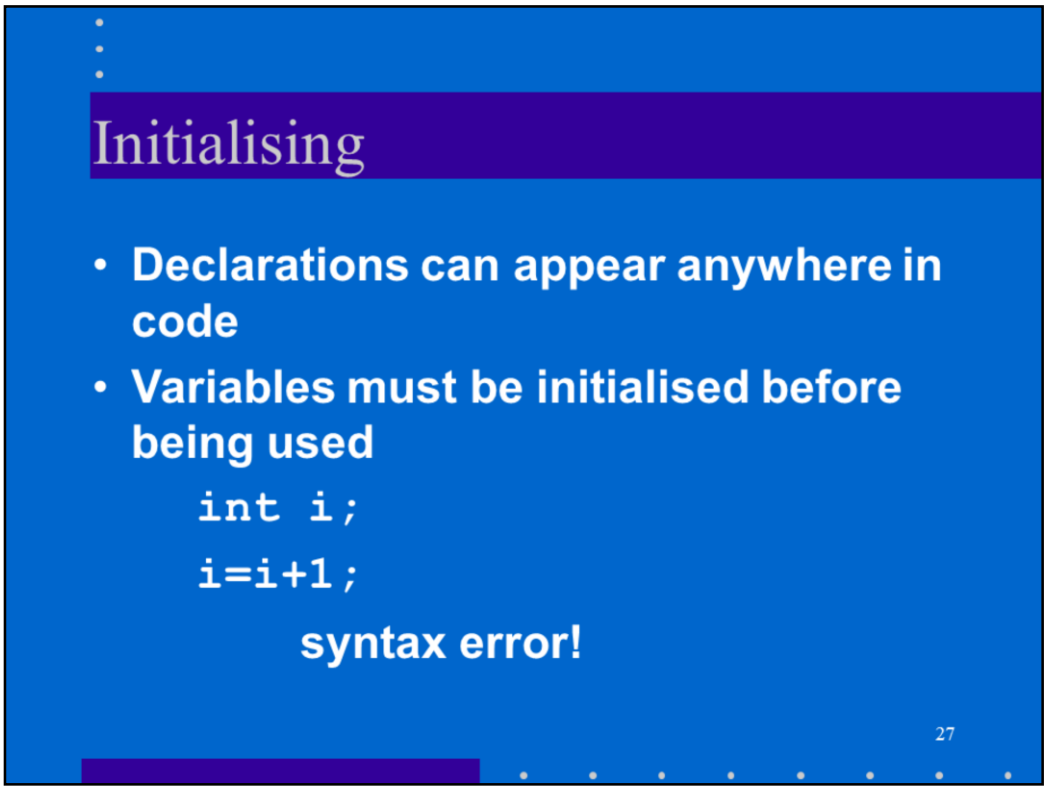
To make programs more readable many Java programmers adopt a simple convention in naming the elements of a program. The first letter in the name of a variable or a method is normally lower case. The first letter of a class name is normally upper case.

Typically Java programmers name things with a string of words e.g. theAreaOfTheCircle. The first character of such a name follows the rules above, if it is the name of a class then it is upper case otherwise lower case. After that the first letter of each word in is uppercase.

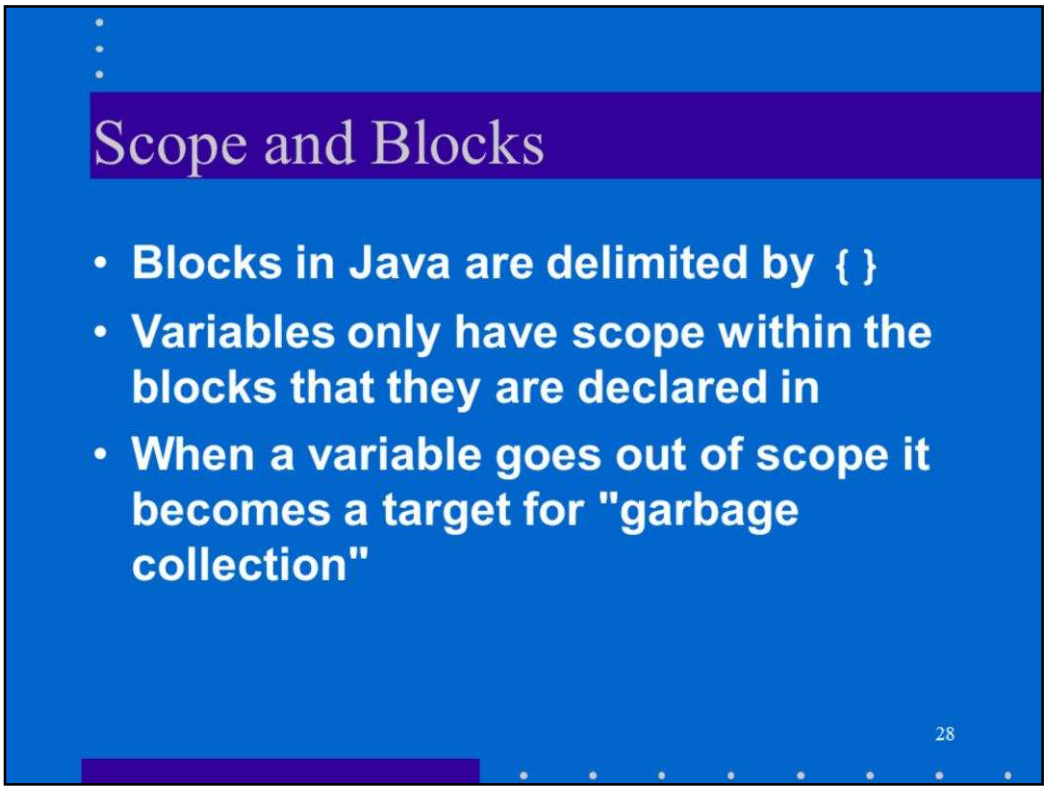


Declarations of variables in Java are made by giving the type name and then a list of variable names for that type. The variable names are separated by commas. As with C and C++ every statement in Java is terminated by a semi-colon.

Variables may be initialised within a declaration statement as shown on the right hand part of the slide.

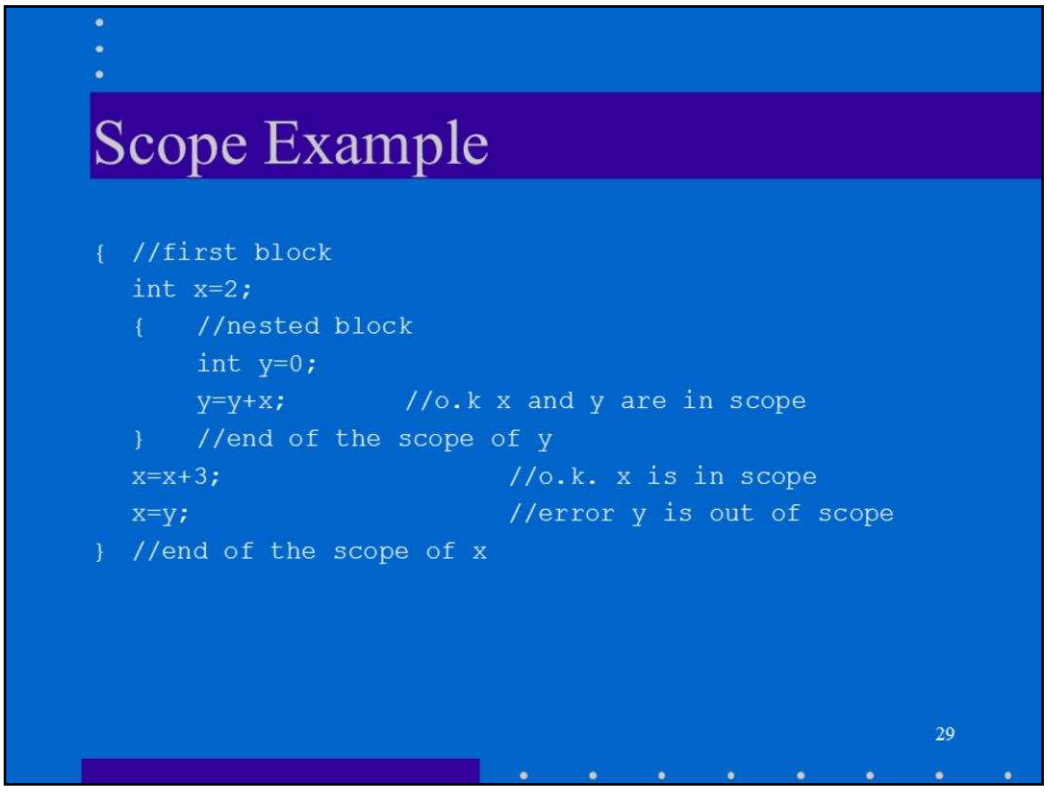


As with C++ (but not C) a variable can be declared anywhere in the program code. One difference between a Java compiler and a C++ compiler, however, is that Java will not let you use a variable unless it has been assigned a value. In C++ the compiler might generate a warning if you try this. In Java a syntax error will be generated and the compilation will fail. This is an example of the way Java attempts to make sure that programs are safe as the use of an uninitialised variable can be perfectly acceptable on some machine architectures but disastrous on others.


Scope and Blocks
• Blocks in Java are delimited by { }
• Variables only have scope within the blocks that they are declared in
• When a variable goes out of scope it becomes a target for "garbage collection"
28

As with C++ and C, Java uses { to begin a block and } to end a block. In Java, a block may be used at any place in a program where a statement may appear. This concept is useful when a programmer wants to execute a number of statements. For example, when the condition of an `if` statement evaluates as `true` and in many other places in a Java program.

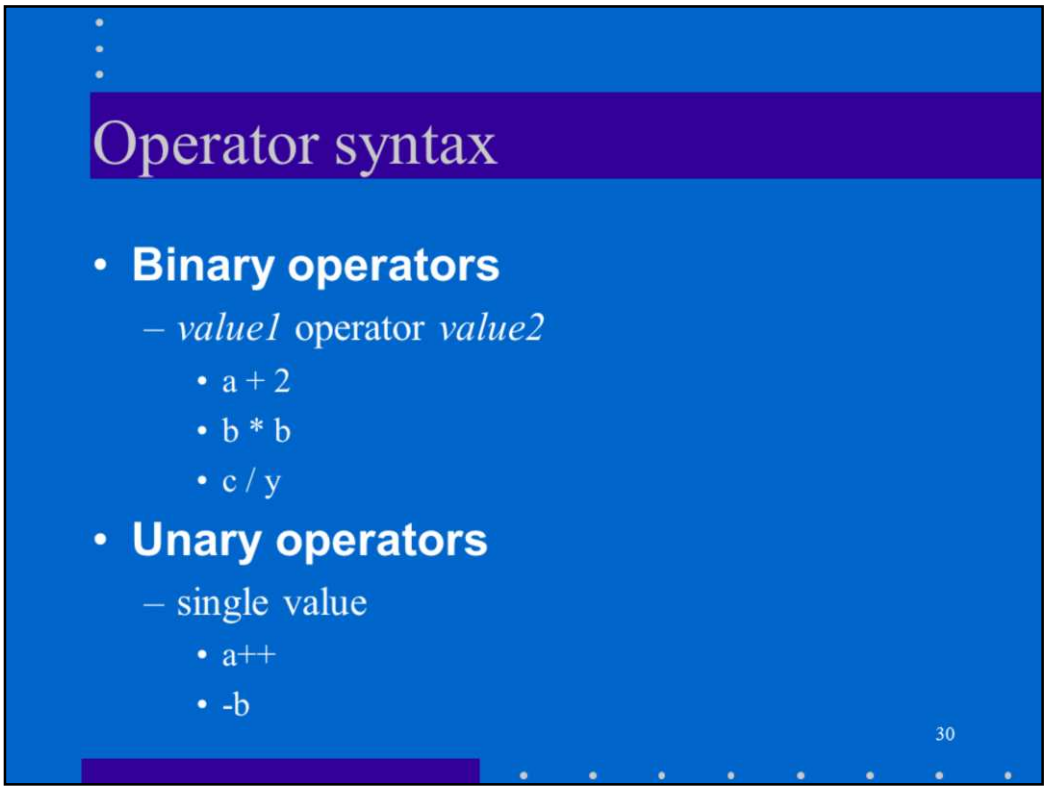
If a variable is declared within a block it can only be used within that block and within any blocks nested in that block. It will cease to be available when control leaves that block.


Scope Example
{ //first block
int x=2;
{ //nested block
int y=0;
y=y+x; //o.k x and y are in scope
} //end of the scope of y
x=x+3; //o.k. x is in scope
x=y; //error y is out of scope
} //end of the scope of x
29

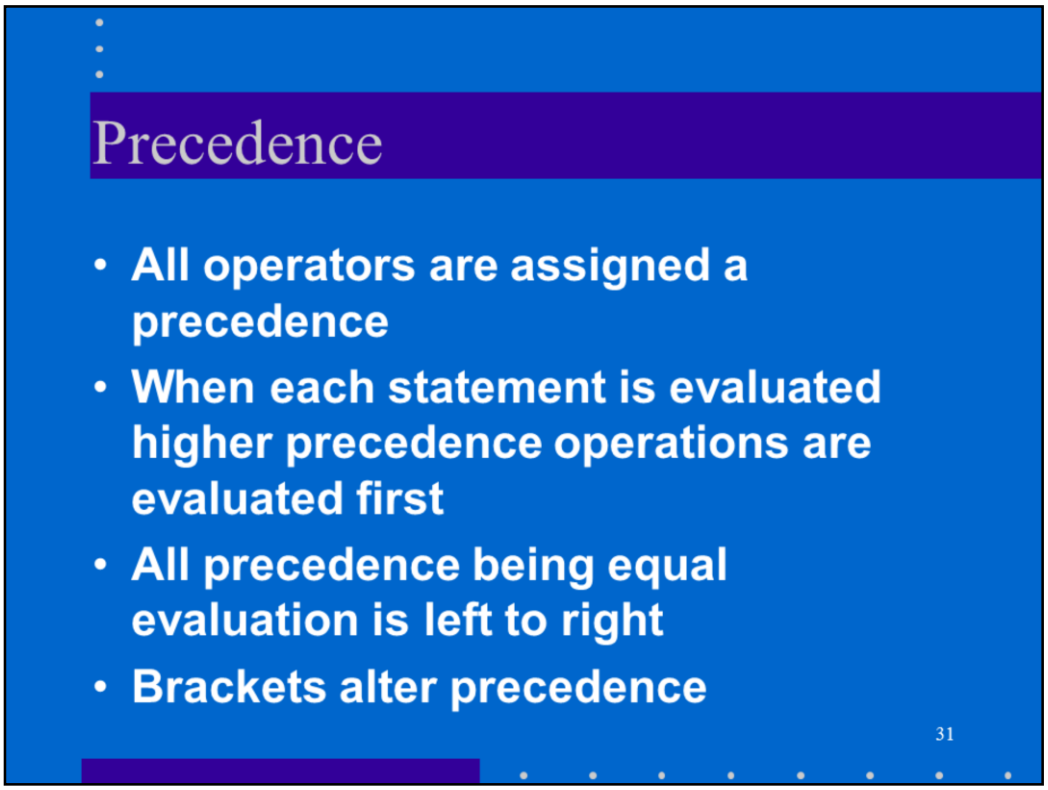
All statements in Java other than declarations appear in blocks of code known as *methods*.

A variable name cannot be used twice in the same method even if its declaration appears in separate blocks.

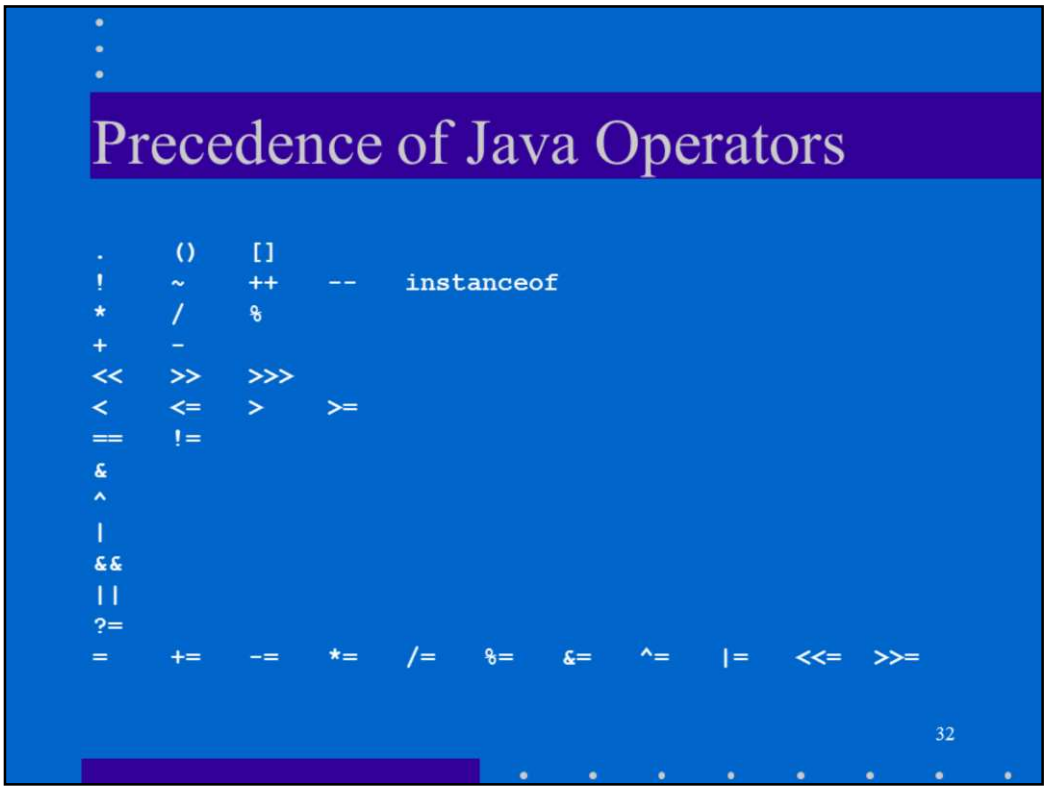
During the execution of a program, when a variable goes out of scope Java re-uses its storage via a mechanism known as garbage collection. Automatic garbage collection is one of Java's special features. This feature can be contrasted with the mechanism used in C and C++ where the programmer is responsible for allocation and deallocation of memory. A large percentage of the code in C and C++ is dedicated to memory management.



Java's operator syntax distinguishes between operators that require two values such as addition and multiplication and those that require a single value such as negation. Operators requiring two values are *binary operators*, operators requiring a single value are *unary operators*.


Precedence
• All operators are assigned a precedence
• When each statement is evaluated higher precedence operations are evaluated first
• All precedence being equal evaluation is left to right
• Brackets alter precedence
31

In order to evaluate an expression it is necessary to establish the order in which operations within the statement are carried out. In Java as in C and C++ some operators will be evaluated before others. Operators which are evaluated first have a higher *precedence*. If all the operators have the same precedence then the expression is evaluated from left to right. Brackets can be used to override precedence in the same way as C and C++.



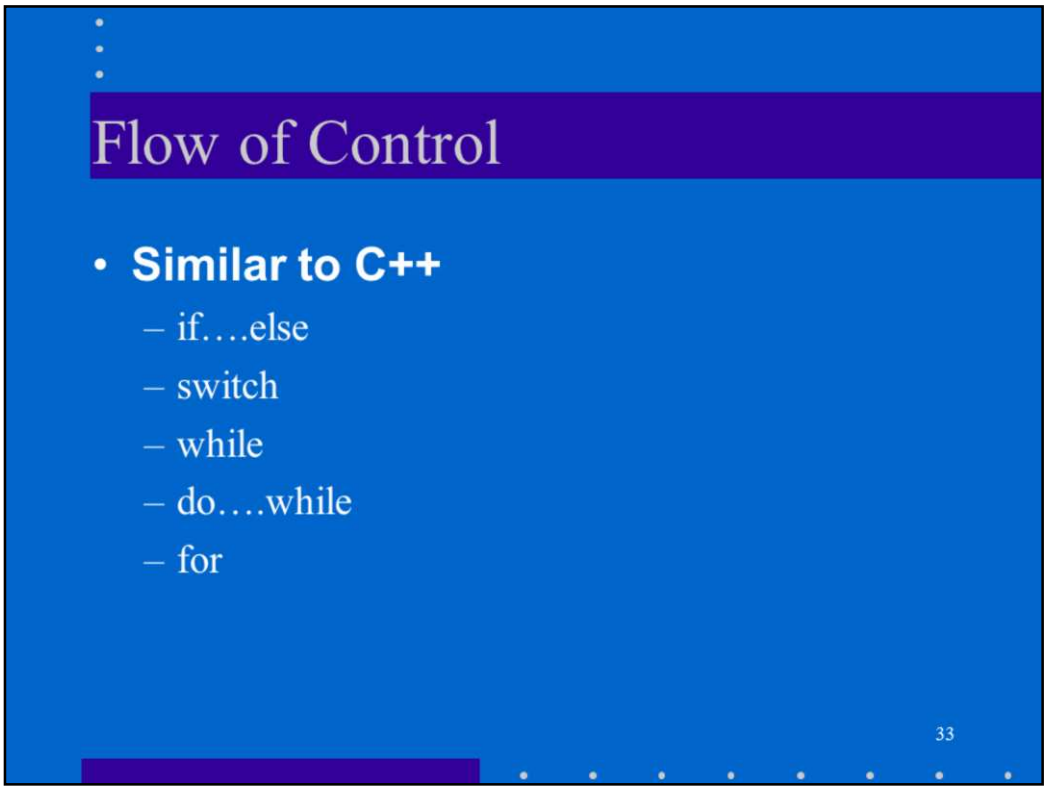
A full description of all the operators in Java can be found a t
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html

3+4*5 evaluates as 23 not 35

(3+4)*5 evaluates as 35

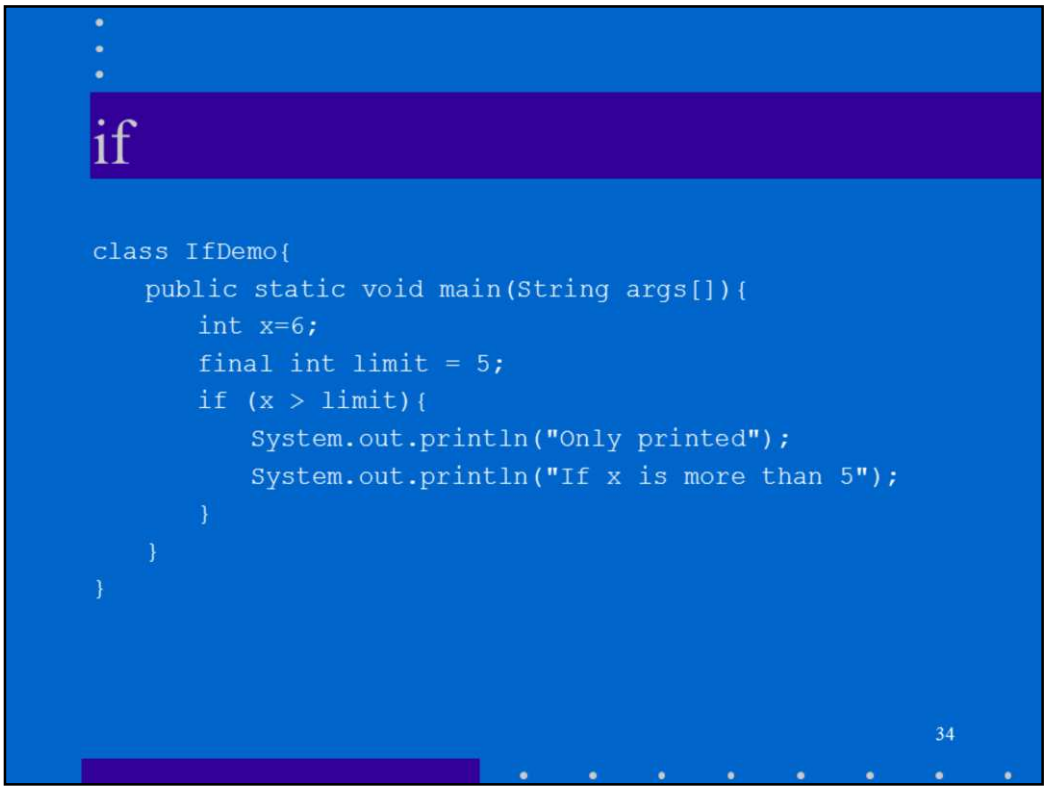
3-4*5 evaluates as -17 not -5

(3-4)*5 evaluates as -5



The normal flow of control in a Java routine is sequential i.e. one statement following after the next. The slide shows the statements in Java which can be used to alter this flow of control. `if….else` and `switch` can be used to make decision branches. `while`, `do….while` and `for` are used to construct loops.

The use of these statements in Java is almost identical to their use in C++. Seen the next slides for examples.



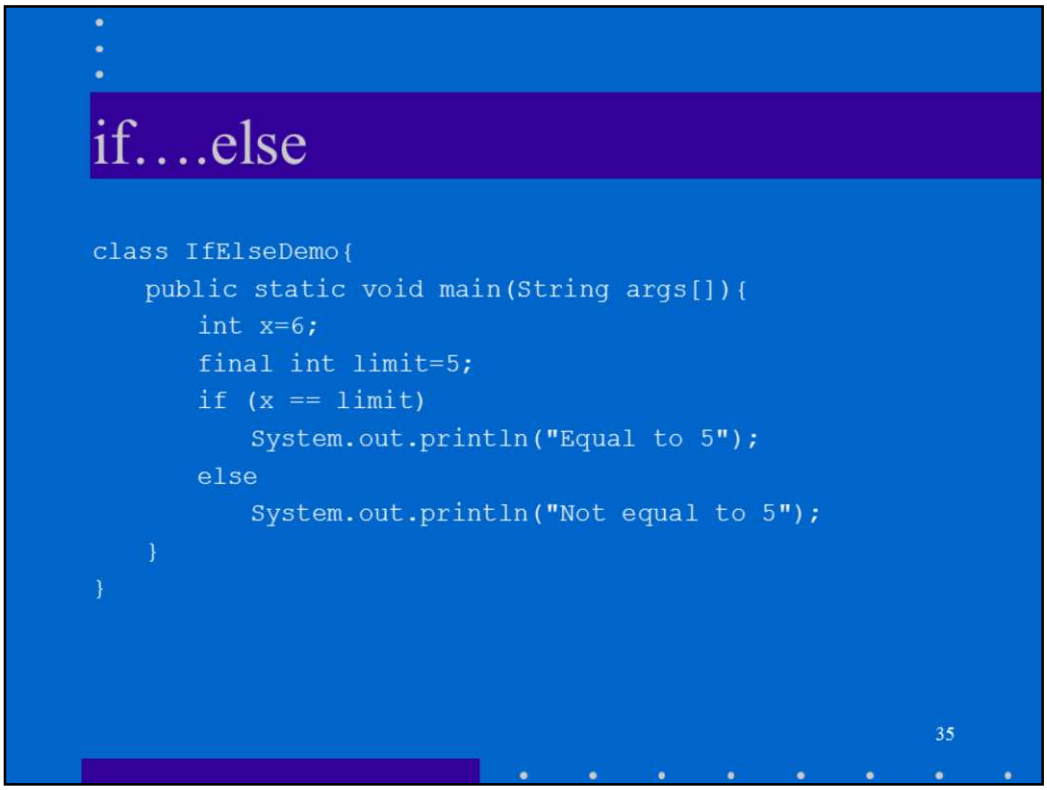
The basic form of `if` is:

`if` (*test-expression*) *statement*

The program shows a test on a variable `x` where some text is output if the value of `x` is greater than 5.

Note that this is an application with a main method. As with all Java programs it has to be written as a `class`.

In this case a block has been used in place of a statement.



The general form of if…else is:

`if` (*test-expression*) *true-statement* `else` *false-statement*

Here an `else` part has been used in the `if` statement. The program will print out a suitable message if `x` is equal to 5 and a different message if `x` is not equal to 5.

Single statements have been used in this program but blocks may be used instead.

switch

```
class SwitchDemo{
    public static void main(String args[]){
        int choose=2;
        switch(choose){
            case 1:
                System.out.println("Choice 1");
                break;
            case 2:
                System.out.println("Choice 2");
                break;
            default:
                System.out.println("Fail safe");
                break;
        }
    }
}
```

36

The `switch` statement performs the same function as a case statement in languages such as Pascal. It is identical to the switch statement in C++.

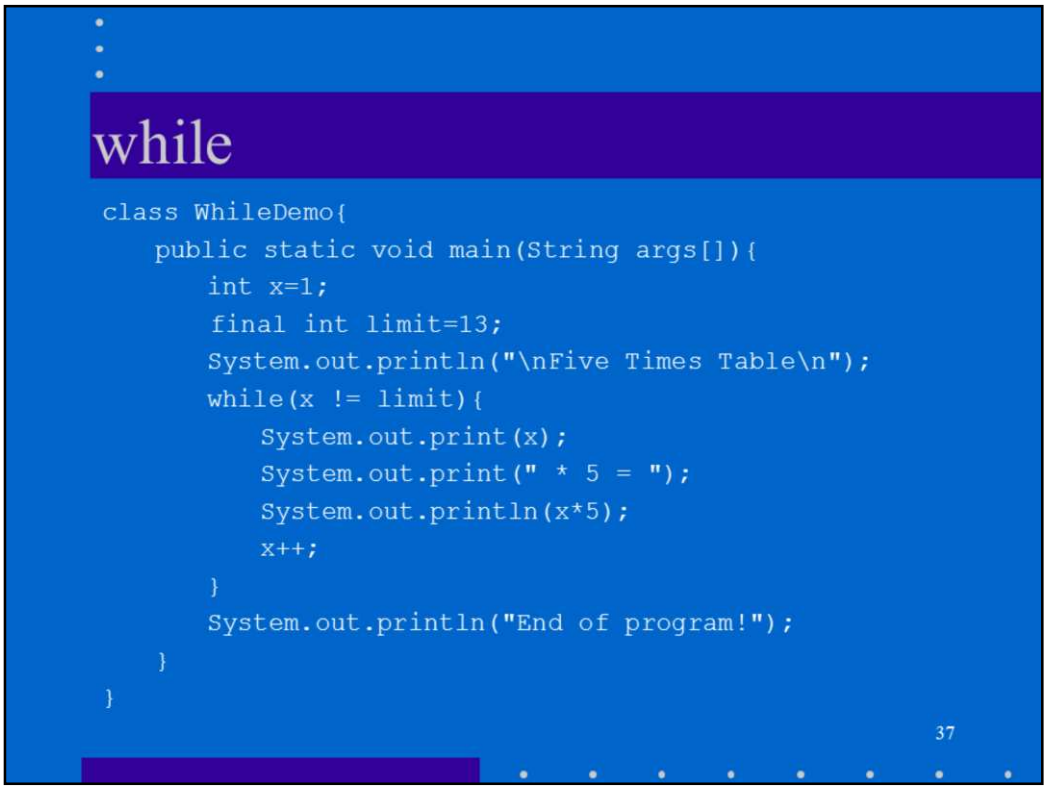
The format is

`switch` (*case-expression*)`{`
`case` *literal-value*`:` *statements*
`case` *literal-value*`:` *statements*
`default:` *statements*
`}`

As Java treats integer and characters equivalently it is possible to use characters for the literal values in `switch` statements.

The *case-expression* must evaluate to an integer value. If the integer value matches any of the *literal-values* following the `case` keyword then the statements following that *literal-value* are executed. If there is no match the the statements following `default:` are executed. It is common practice to end a list of statements in a `switch` statement with a `break` statement. This moves the flow of execution to the first statement after the `switch` statement. If `break` is omitted then all the statements in the cases following the selected one will be executed (there are times when this is useful!).

The program above will output the string "Choice 2".



This is identical to the while loop in C++.

The format is:

`while (`*test-expression*`)` *statement*

test-expression must evaluate to true or false. The statement or block of statements is executed as long as the *test-expression* evaluates to true.

If the *test-expression* evaluates false the first time it is encountered then the loop is not entered.

The program on the slide calculates the five times table from one times five until 12 times 5. The loop control variable is incremented until it equals 13 and then the program exits from the loop.

This program uses a new method of the object `System.out`. The method `println` always appends a new line to output whereas `print` does not.

Note also the use of the `\n` escape character to generate additional new lines.

do..while

```
class DoWhileDemo{
    public static void main (String args[]){
        int x=1;
        final int limit=12;
        System.out.println("\nFive Times Table\n");
        do{
            System.out.print(x);
            System.out.print(" * 5 = ");
            System.out.println(x*5);
            x++;
        } while (x<=limit);
        System.out.println("End of Program");
    }
}
```

38

The general format of `do..while` is:

`do` *statement* `while (`*test-expression*`);`

As in `while`, *test-expression* must evaluate to true or false.

The statement or block of statements is **always executed once** and execution continues until the *test-expression* evaluates to false.

The program shown uses `do..while` to produce the same results as the previous example.

for

```
class ForDemo{
    public static void main (String args[]){
        int x;
        final int limit=13;
        System.out.println("\nFive Times Table\n");
        for(x=1;x<limit;x++){
            System.out.print(x);
            System.out.print(" * 5 = ");
            System.out.println(x*5);
        }
        System.out.println("End of Program");
    }
}
```

39

The for statement is identical to the for statement in C and C++.

The general format is:

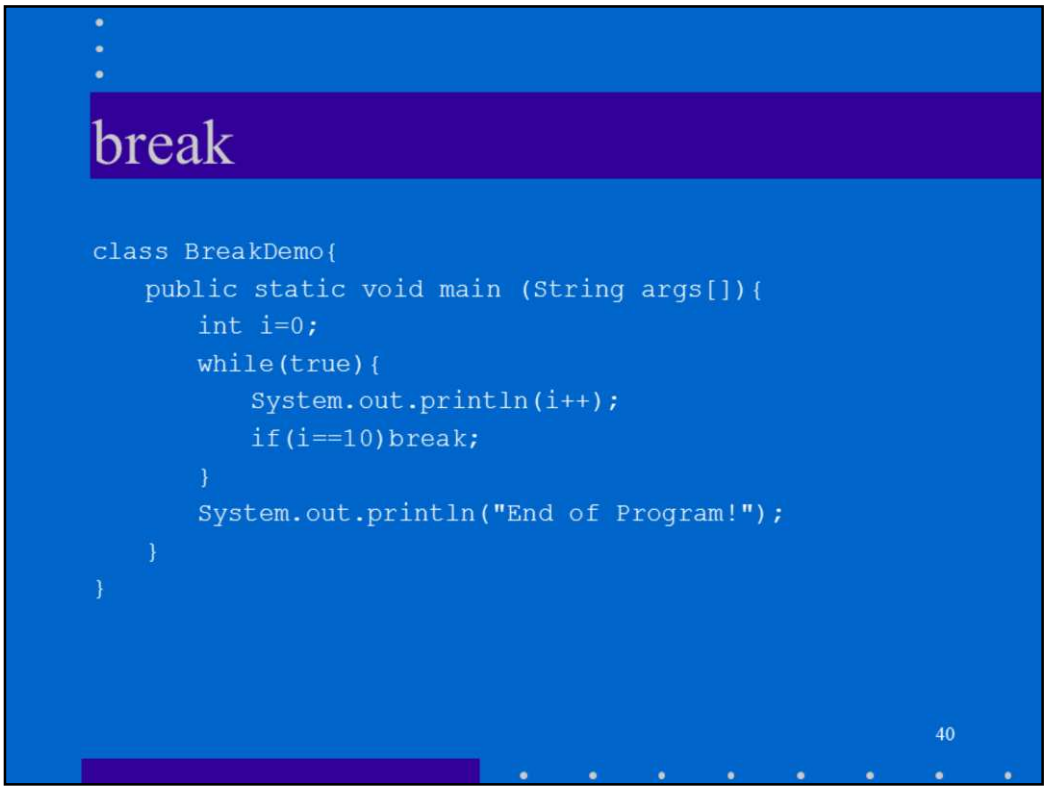
`for` (*initialisation-statement*`;` *test-expression*`;` *modification-statement*)
statement

The *initialisation-statement* normally gives the loop variable an initial value.

The *test-expression* must evaluate to `true` or `false`. The statement or block of statements in the loop are evaluated while the *test-expression* is `true`. If the initial value of the *test-expression* is `false` then the loop is not entered.

The *modification-statement* performs an operation after each execution of the loop. Typically this will be a modification of the loop variable.

The program shown sets `x` to an initial value of 1 and increments `x` by 1 after each execution of the block of statements in the loop. This continues as long as `x` is less than 13. This achieves the same result as the previous examples of loops.

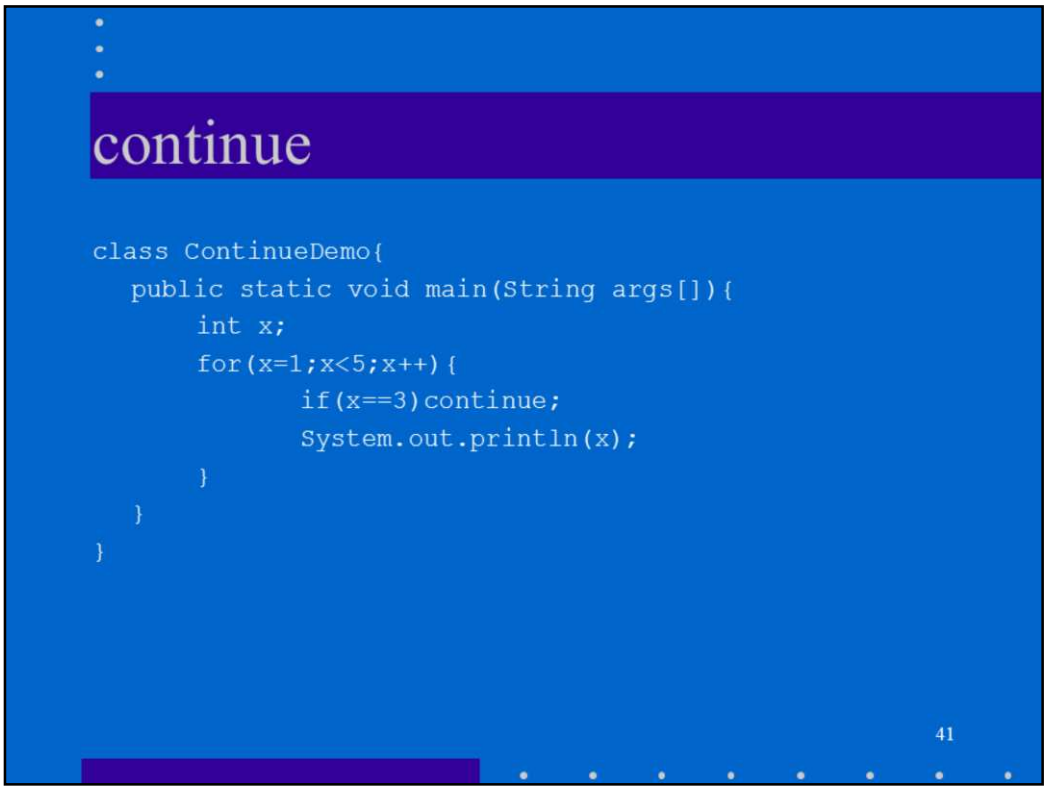


`Break` is used to move the flow of control to the statement following the current construct (loop or `switch` statement).

The `while` loop in the example will execute forever because the *test-expression* will always evaluate to true. Flow of control moves out of the loop, however, when `i` is equal to 10 because a `break` statement is executed.

The program will display the numbers 0 to 10 and then terminate.

Some people would recommend that `break` is never used in a program and that appropriate `if` statements should be used instead.

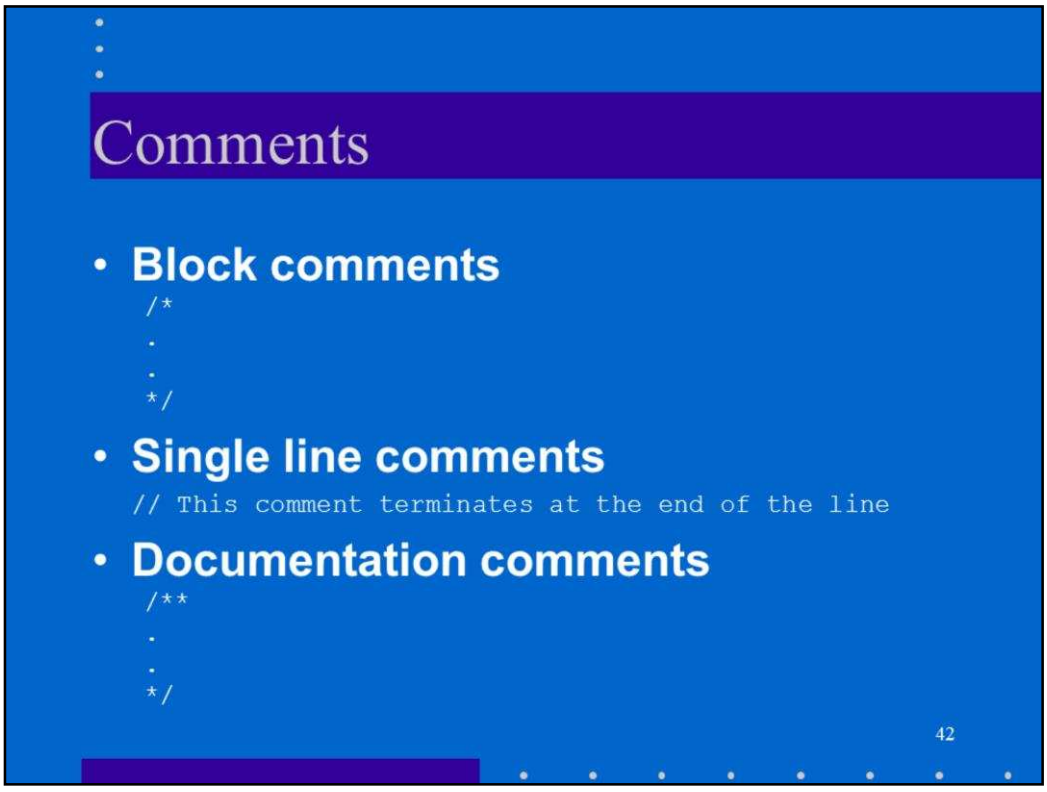


The `continue` statement moves the flow of control on to the next iteration of a loop before the final statement in the loop has been reached.

The loop above generates the values 1 to 4 for x. When x is equal to 3, however, a `continue` statement is executed. This moves the flow of control to the top of the loop and the println method is not executed. So instead of printing the numbers 1 to 4, the program prints 1, 2 and 4.

The same objections raised against `break` also apply to `continue`. Again, `continue` can be made unnecessary by correct use of `if` constructs.

`break` and `continue` in Java are similar to their counterparts in C and C++. Java, however, adds one extra feature. `break` and `continue` in Java may specify a label. This enables a programmer to select which construct to return to. This feature is probably best avoided as it may make code prone to errors.



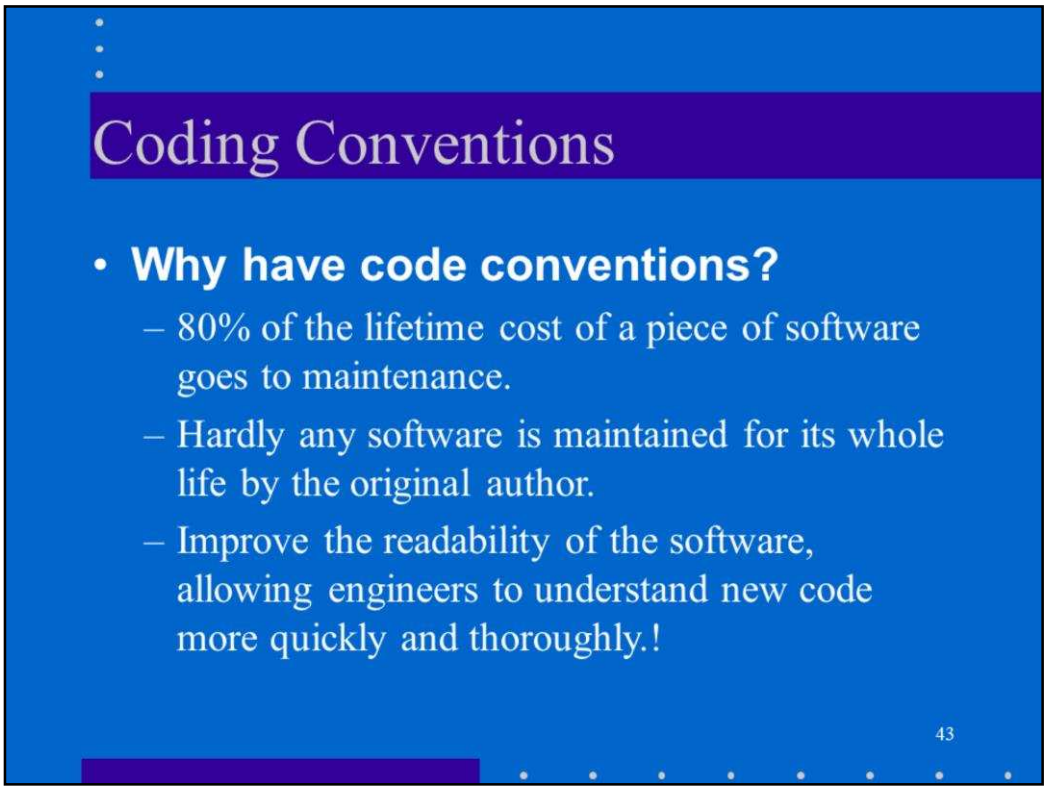
Java supports three types of comments. Two of these are identical to the comment syntax found in C++

A comment which spans several lines begins with /* and terminates with */. All the lines between /* and */ are ignored by the compiler.

Single line comments are indicated by //. All the characters following the // are ignored until the end of the line. // may appear at any position on a line so you can add a comment to a statement e.g.

```
i++; // increment loop counter
```

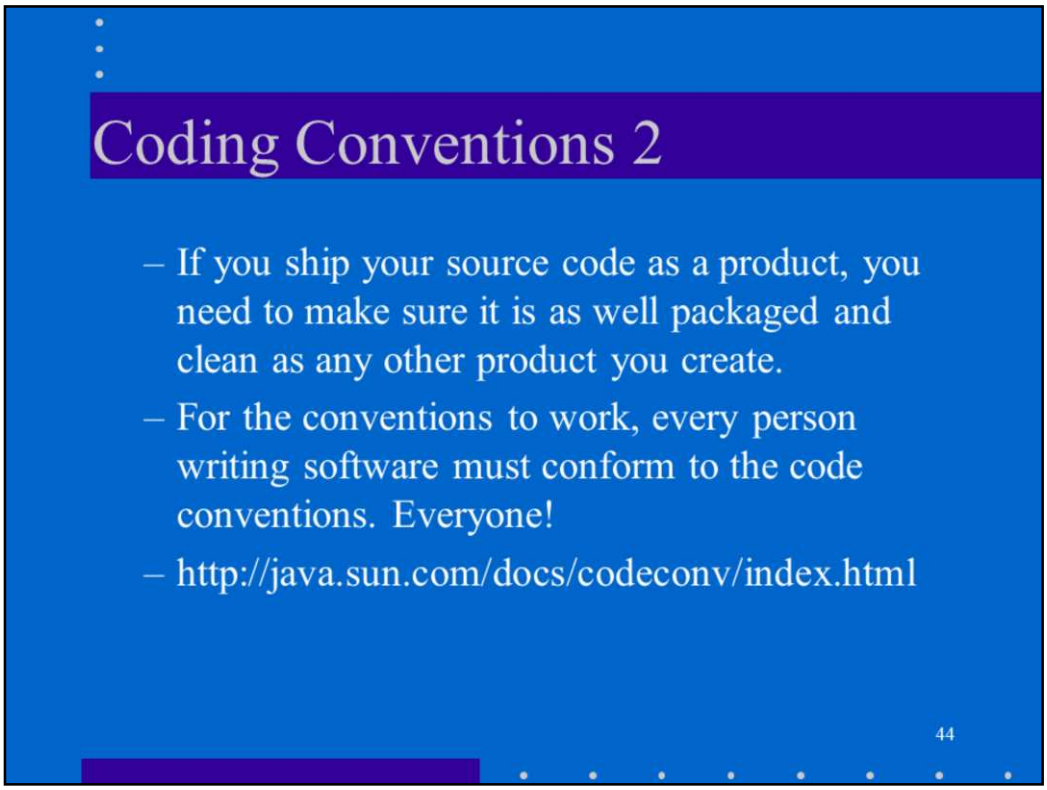
Documentation comments look a bit like block comments but begin with /**. Java documentation can be generated from source code as you will see in one of the workshops. It is possible to generate rudimentary documentation from an uncommented program but documentation comments are necessary in order to create documentation similar to that provided in the Java Development Kit (JDK).


Coding Conventions
• Why have code conventions?
– 80% of the lifetime cost of a piece of software goes to maintenance.
– Hardly any software is maintained for its whole life by the original author.
– Improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.!
43

See:

Vemeulen et al. “Elements of Java Style” Cambridge University Press, 2000, ISBN 0-521-777682

http://java.sun.com/docs/codeconv/index.html


Coding Conventions 2
– If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
– For the conventions to work, every person writing software must conform to the code conventions. Everyone!
– http://java.sun.com/docs/codeconv/index.html
44

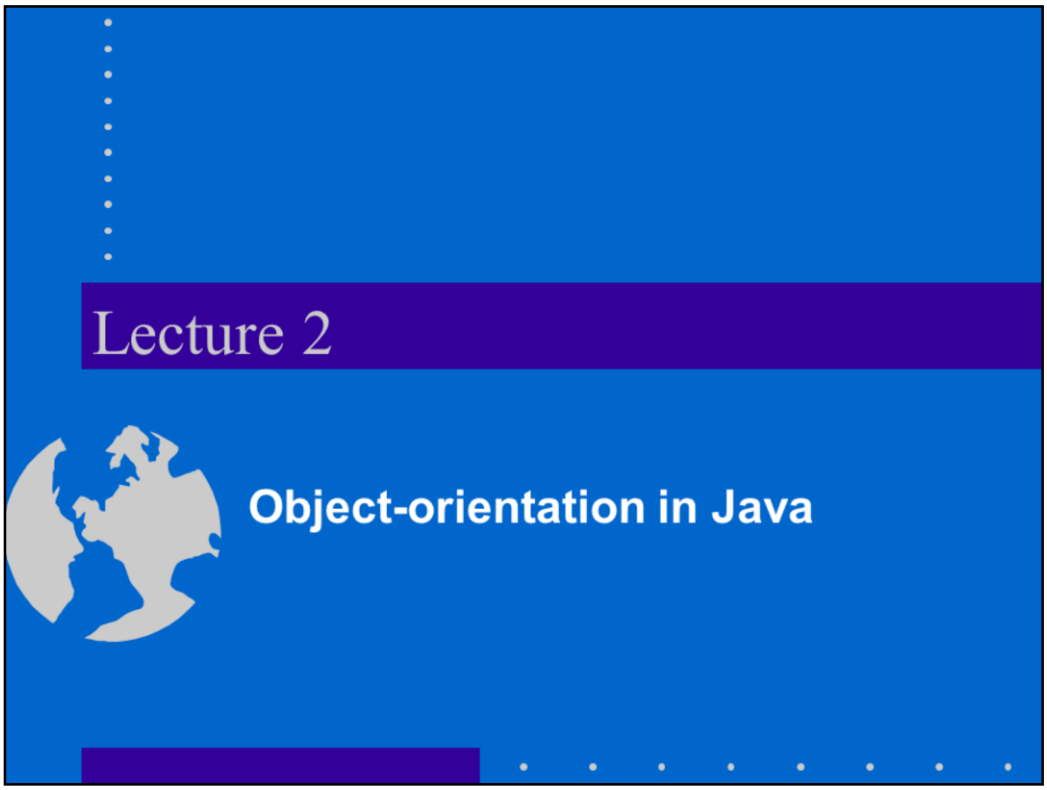
See:

Vemeulen et al. “Elements of Java Style” Cambridge University Press, 2000, ISBN 0-521-777682

http://java.sun.com/docs/codeconv/index.html

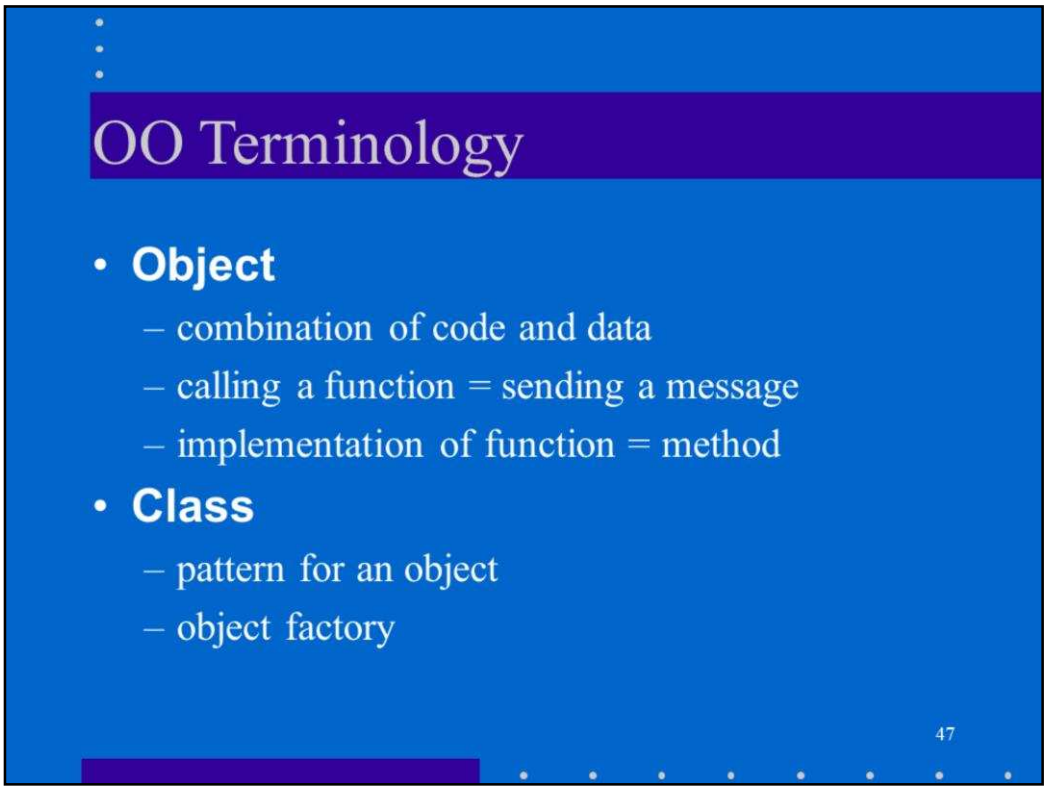
Summary

- Background to Java
- Literals in Java
- Basic types in Java
- Selection in Java
- Iteration in Java
- Comments



Java is an object-oriented language. A Java programmer must grasp the essential elements of object-orientation in order to begin to write a program. Whilst C++ can be used as an object-oriented language the necessity for backward compatibility with C means that it can also be used as a conventional procedural language.

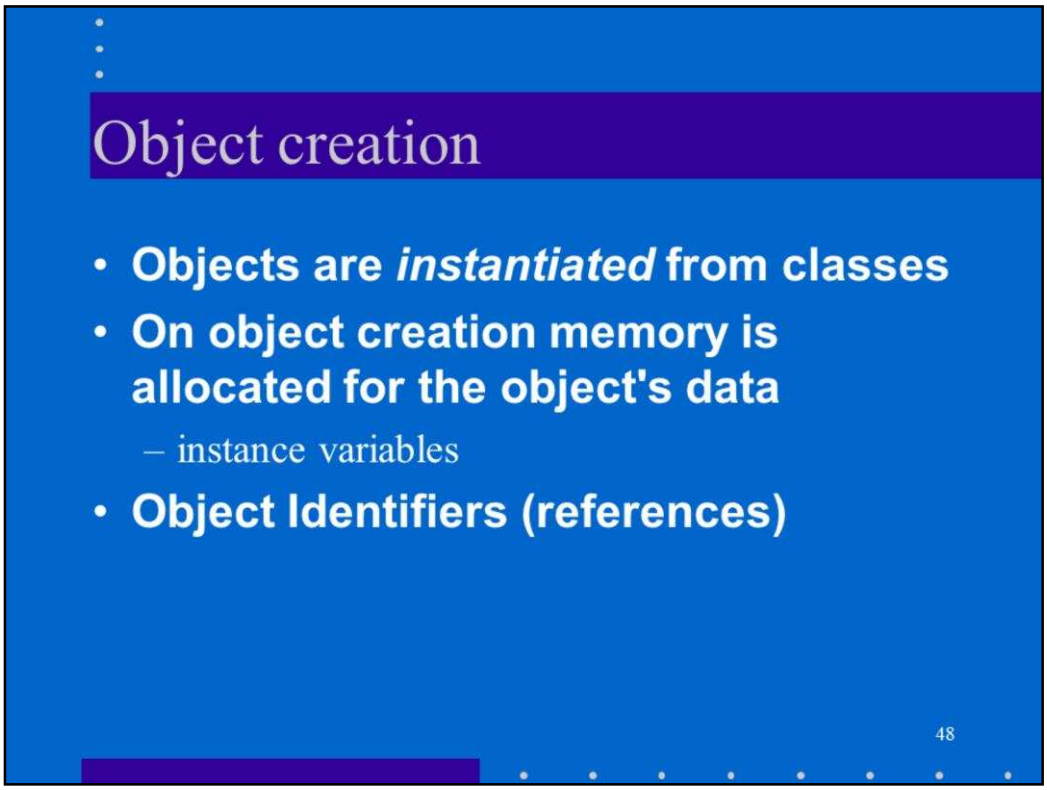
At the time Java was designed there was over twenty years of experience of object-oriented programming available to its designers. This meant that Java was able to pick and choose the best bits from a variety of object-oriented languages. Java has the syntax of C++ but the feel of Smalltalk. However, it sacrifices some of the object purity of Smalltalk for a more pragmatic approach.



Objects are mechanisms which help us think about programs. Most programs manipulate data. Given any collection of data, there is a finite set of things we can sensibly do to that data. The make up of the operations in this set will depend on the real world entity that the data represents. For example if we have a value representing the number of items in a fruit bowl then whilst we might want to implement a subtraction operation, that operation should never produce a negative result as it isn't possible to have a fruit bowl with less than zero items of fruit in it. On the other hand if the value represents a bank balance then subtraction can produce a negative value since, sadly, negative bank balances do often occur.

An object is simply a way of collecting together data and the operations on that data. In conventional object-oriented terminology the operations are called methods.

A class is a pattern for a set of objects. Classes define the data which is stored in each object of that class and the methods that each object of the class implement.


Object creation
• Objects are *instantiated* from classes
• On object creation memory is allocated for the object's data
– instance variables
• Object Identifiers (references)
48

Objects are instantiated (created) by operating on classes. That is instances of objects can be created by applying the appropriate operator to a class. Java programs manipulate objects.

A class defines the set of instance variables that each object of that class will have. When an object is created, space in the computer's memory is made available to hold the values of these instance variables.

Every object has a unique identity which is called its object identifier (oid). In C++ an object identifier is effectively a pointer to the memory location of the object. Java does not implement pointers for programmers to use, as pointers appear to be the source of a lot of programmer errors. In Java, when an object is created its oid may be stored in a variable as an object reference. Whilst C++ permits pointer arithmetic an object reference supports only a limited set of operations which do not include arithmetic.

Every Java program begins with a class definition. All classes have a name. A class may extend another class and implement a number of interfaces (see inheritance later in the module). A class may contain some instance variable definitions. A class may contain some method declarations.

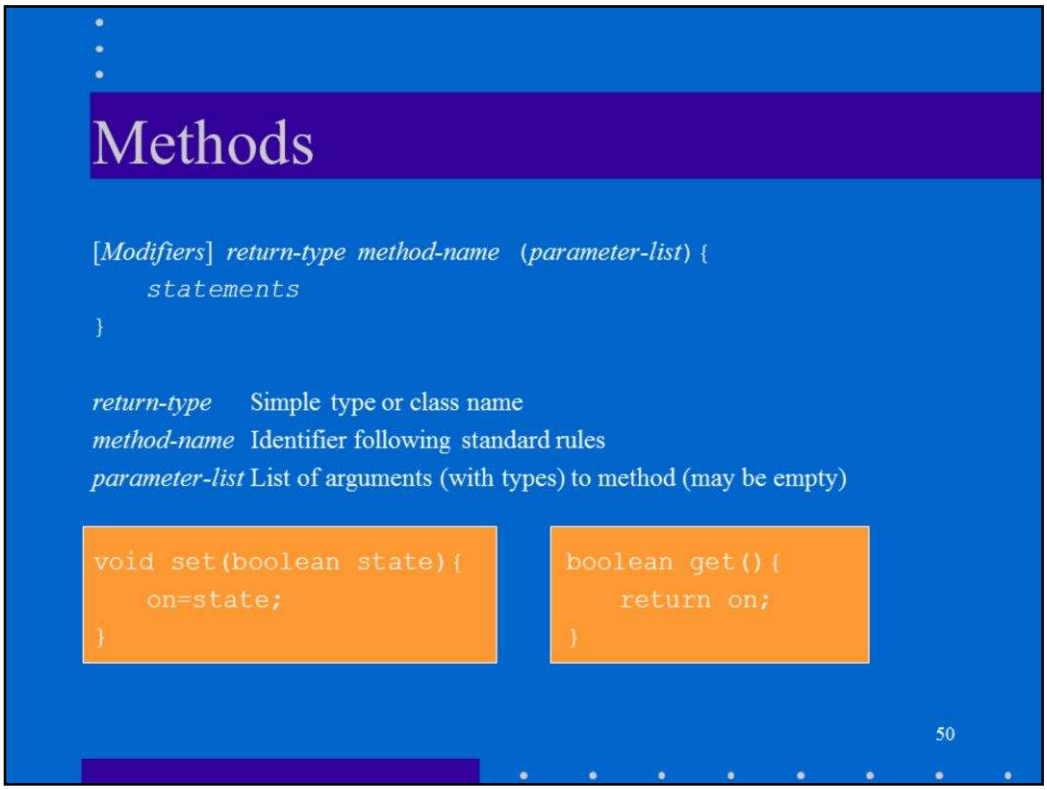
The class on the slide is named `Heater`.

It has one instance variable of type `boolean` called `on`.

The class represents a real world object which is a heater. The only property of the heater that the designer of the program is interested in is whether the heater is on or off. If the heater is on then the instance variable `on` will be set to `true`. If the heater is off then `on` will be set to `false`.

The variable `on` has an initial value of `true (which is set when the variable is declared)`. That means that a newly created heater will be deemed to be on.

get() is a method which returns the value of on (which is type boolean).



Methods are declared in the way shown on the slide.

Modifiers will be described later in the module. They are not needed for the simple examples shown here (but see `static` methods later in this lecture).

A method may return a value. Every method (except a constructor, see later) must specify the type of the value it returns. If it does not return a value this should be given as `void`. Types can include the simple types covered in lecture 1 or else be the name of a class known to the program.

Every method must have a name.

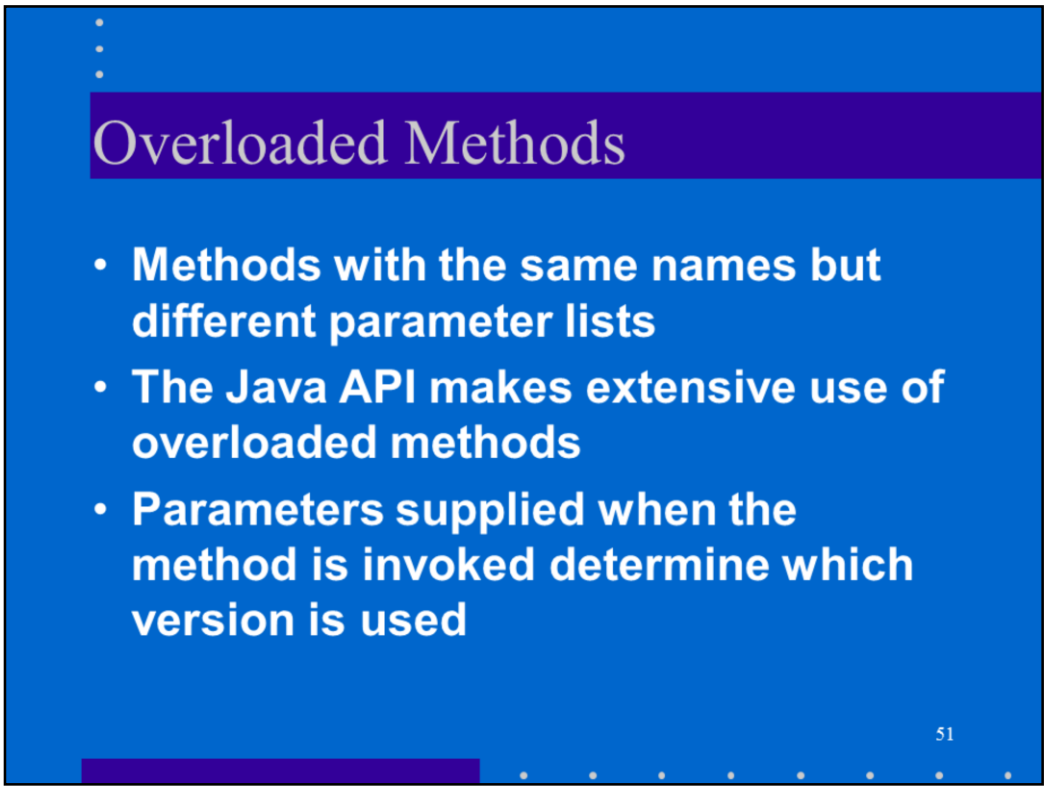
A method specifies the names and types of the parameters it accepts. If it does not accept parameters then this is represented by an empty list `()`. Again the types may be simple types or the names of classes known to the program.

The statements in a method may include declarations of variables local to that method.

There are two methods in the `Heater` class.

`set` does not return a value (hence the use of `void`). It accepts a `boolean` argument and uses this to set the value of the instance variable `on`.

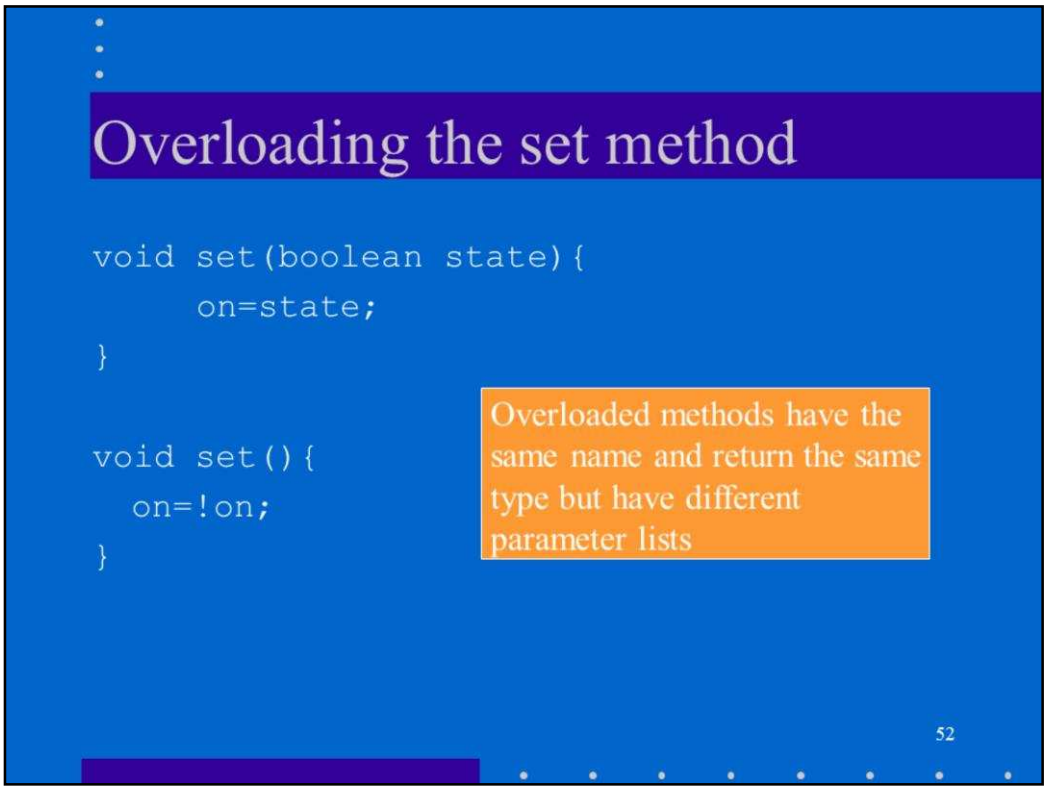
`get` returns a `boolean` value. It simply returns the value of the instance variable `on`.


Overloaded Methods
• Methods with the same names but different parameter lists
• The Java API makes extensive use of overloaded methods
• Parameters supplied when the method is invoked determine which version is used
51

Within a class two methods may have the same name. If this occurs the two methods must define different parameter lists. This is known as *method overloading*. It is a technique used extensively in object-oriented programming and is supported in C++ and VB.Net though not in C.

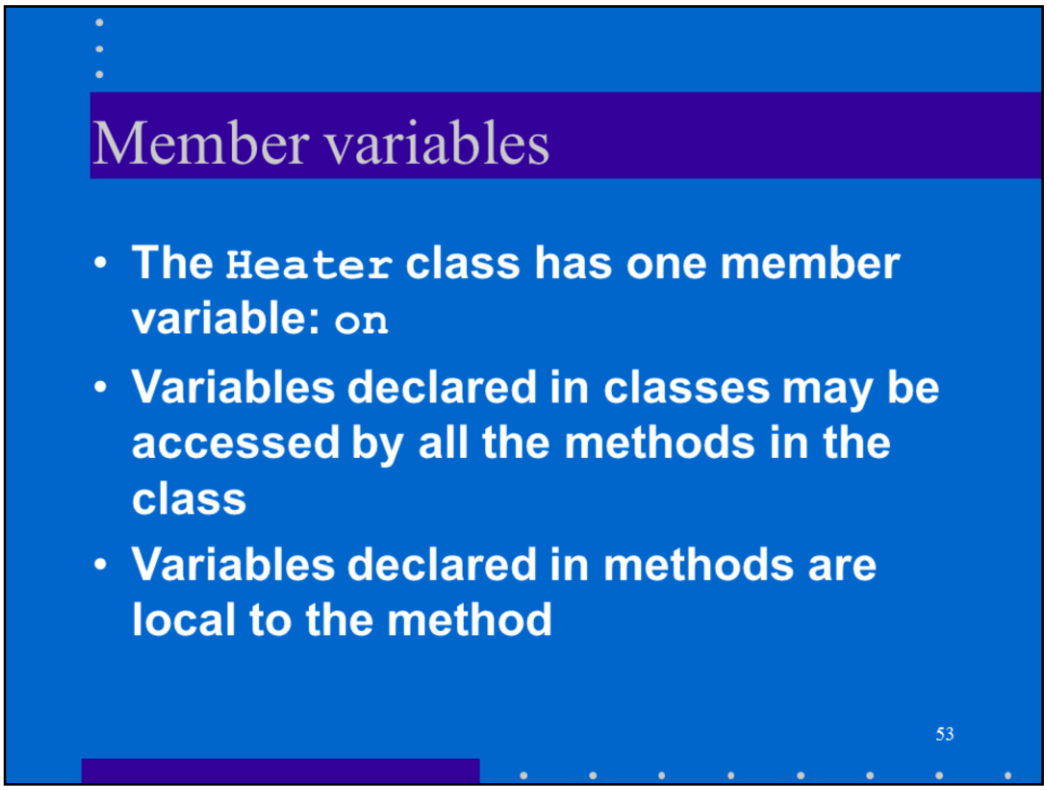
When the method name is used Java will decide which piece of code to use based on the parameters supplied when the method is invoked.

API is an abbreviation for Application Programmer's Interface. This can mean a library of functions and procedures. In Java terms it is a collection of classes.




```
class Heater2{
          boolean on=true;
          void set(boolean state){
                    on=state;
          }
          void set(){
                    on=!on;
          }
          boolean get(){
                    return on;
          }
}
```

In the code above the method called `set` has been overloaded. The new version simply negates the current value of `on`, i.e. if `on` is `true` then it will be set to `false`, if `on` is `false` it will be set to `true`. If `set` is invoked with a `boolean` parameter then the first version is executed. If it is invoked without a parameter then the second version is used.


Member variables
• The Heater class has one member variable: on
• Variables declared in classes may be accessed by all the methods in the class
• Variables declared in methods are local to the method
53

Classes may have any number of member (instance) variables of any types. For example:

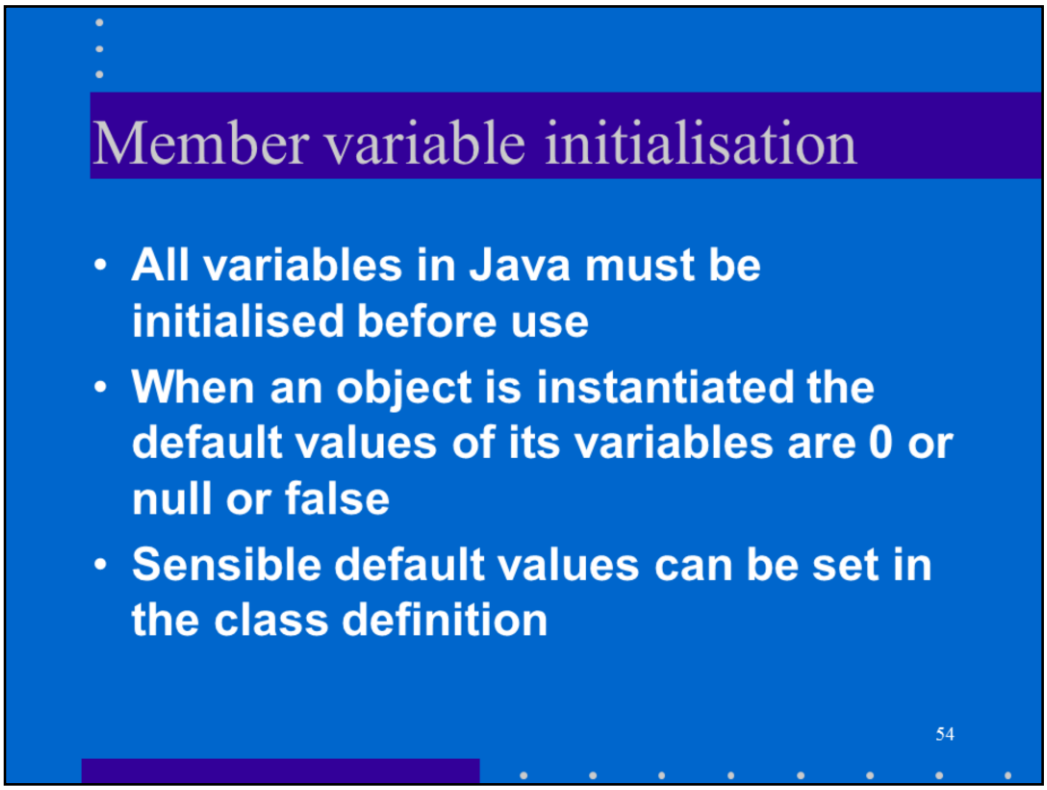
```
class Anyclass{
      int x;
      float y;
      char a,b;
}
```

The `Heater` class only has one instance variable because that's all it needed!

There are scoping rules which apply to variables declared in classes.

If they are declared outside a method then they are in scope in all the methods of the class.

If they are declared inside a method the are only in scope within that method.


Member variable initialisation
• All variables in Java must be initialised before use
• When an object is instantiated the default values of its variables are 0 or null or false
• Sensible default values can be set in the class definition
54

An example in lecture one showed that Java will generate a syntax error if a variable is not initialised before it is used.

Java automatically initialises all the instance variables of an object when it is created.

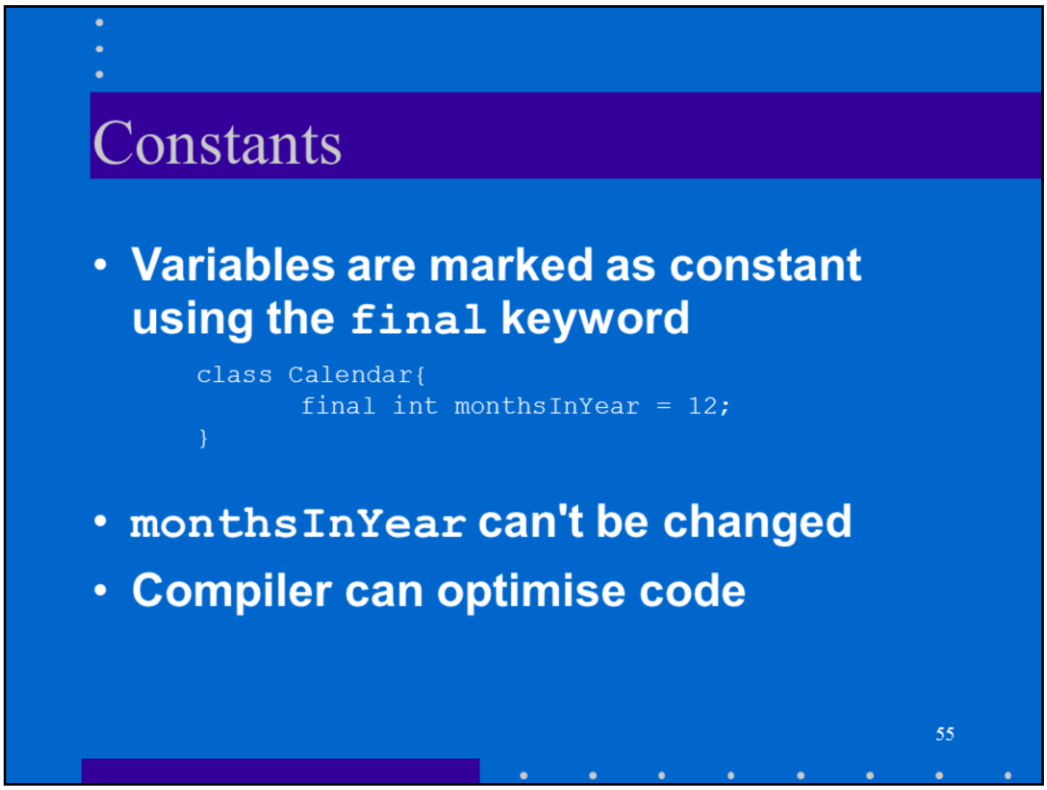
All numeric values (including `chars`) are set to zero.

All boolean variables are set to false.

Everything not numeric or boolena is an object reference and is set to `null` (see later in this lecture).

These may not be sensible values for the instance variables. Instance variables may have a sensible default value specified in their declaration. In the case of `Heater` the single instance variable `on` has a default value of true.

Constructors (see later) may also be used to give default values to instance variables.



Some initialised values should never be changed by the program. For example, the conversion factor between Fahrenheit and centigrade temperatures.

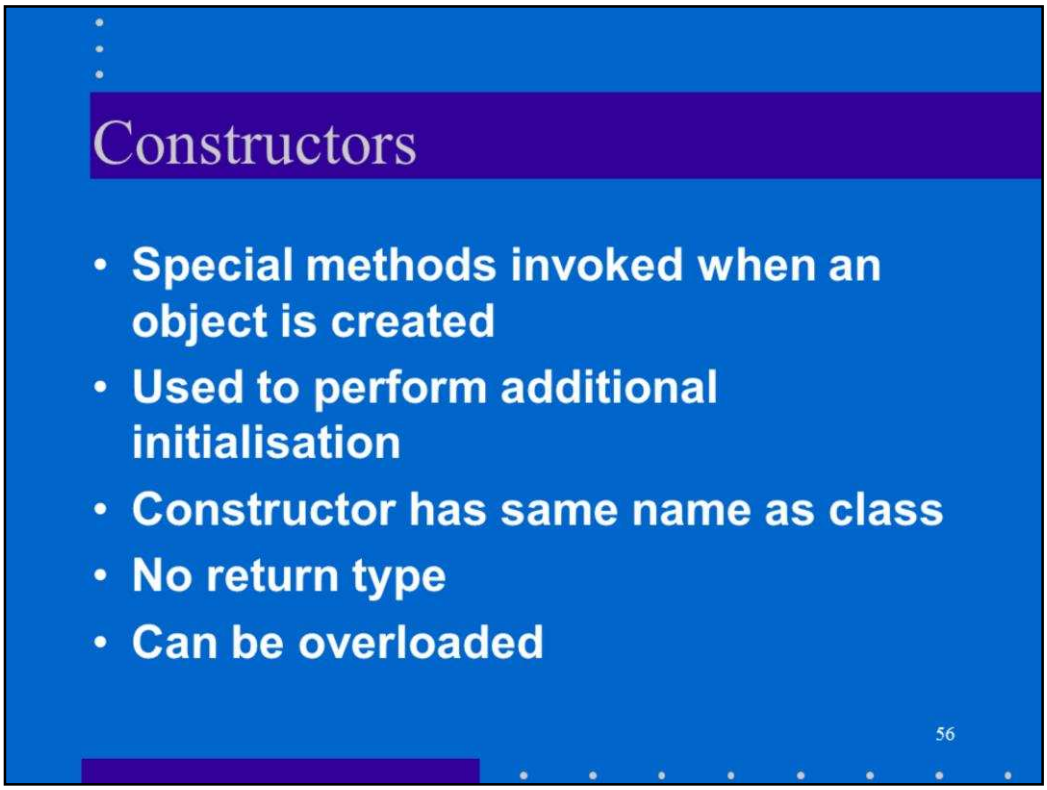
A variable can be made into a constant by prefixing its definition with the keyword `final`. This is similar to const in C++.

An attempt to alter the value of a variable qualified as `final` will produce a compiler error. The code shown below will not compile:

```
class Calendar{
      final int monthsInYear = 12;

      void silly(){
            monthsInYear++;
      }
}
```

The Java compiler can use the fact that a value will remain constant to generate optimised code.

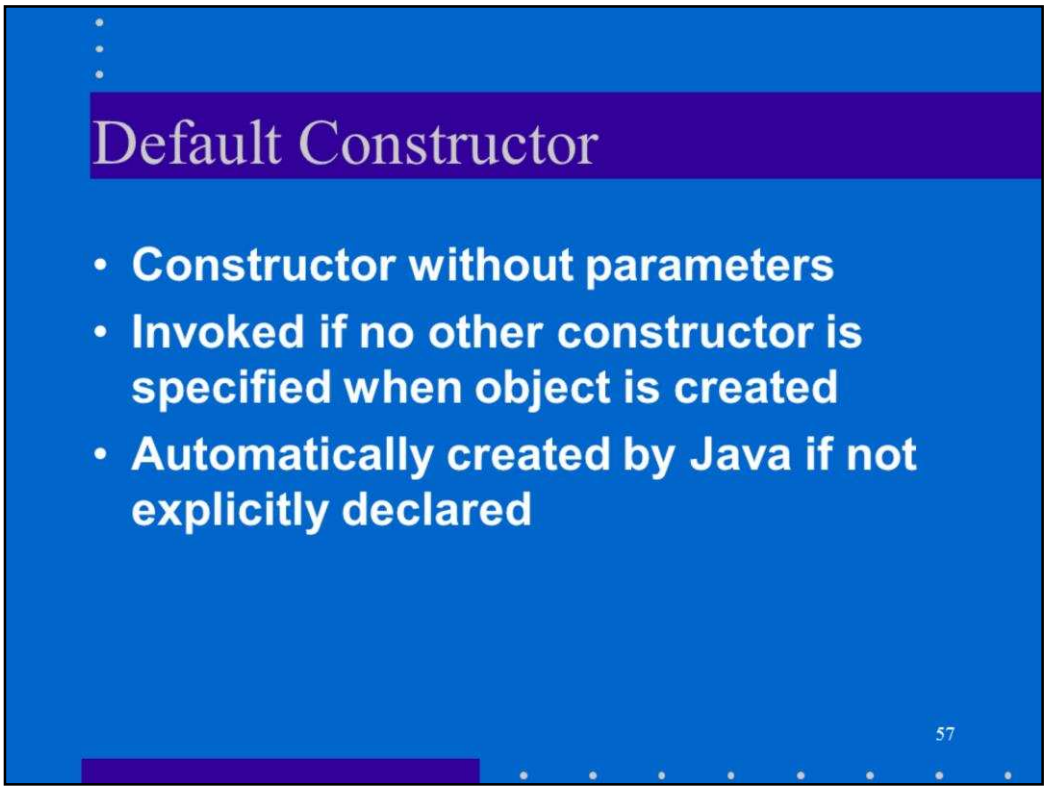

Constructors
• Special methods invoked when an object is created
• Used to perform additional initialisation
• Constructor has same name as class
• No return type
• Can be overloaded
56

Another way in which the instance variables of an object may be initialised is via a *constructor*. In fact a constructor may do much more than just set the values of instance variables.

A constructor is a method but a special type of method. As we shall see later a constructor is always invoked when an object is created.

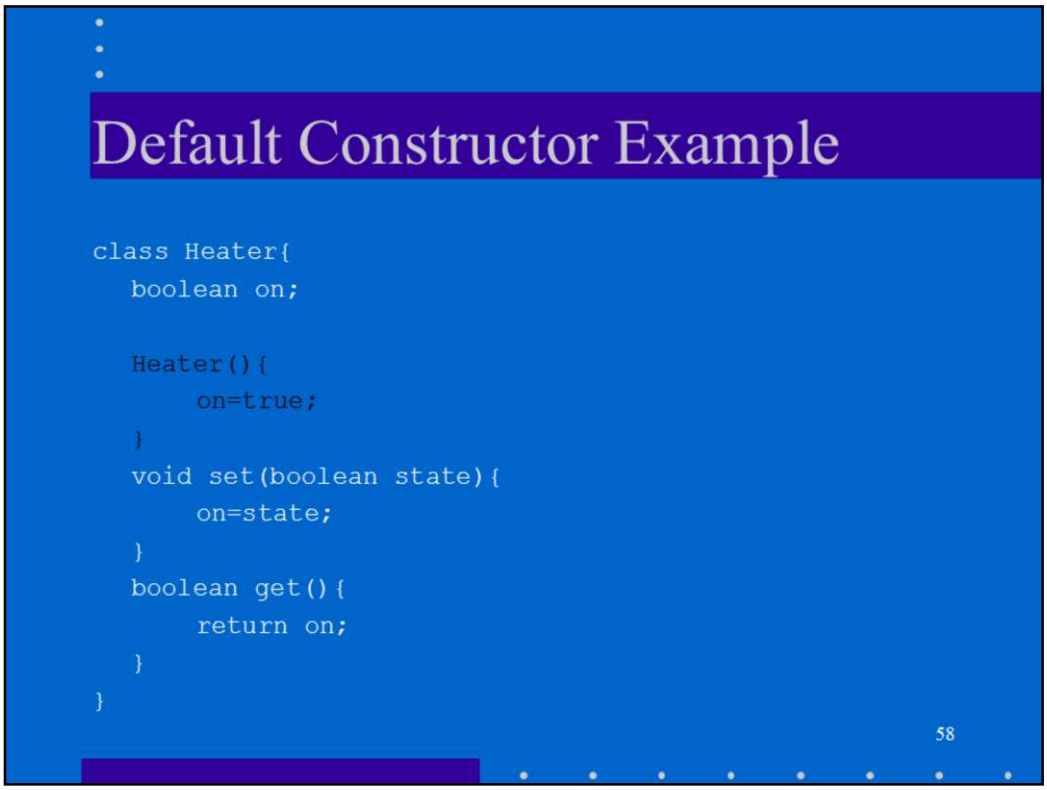
Constructor methods always have the same name as the class they belong to. They do not have a return type (not even `void`).

Constructor methods may be overloaded in the same way as ordinary methods.


Default Constructor
• Constructor without parameters
• Invoked if no other constructor is specified when object is created
• Automatically created by Java if not explicitly declared
57

One special type of constructor is known as the *default constructor.* This is a constructor without any parameters. Every class must have a default constructor and Java will create one if the programmer does not define one.

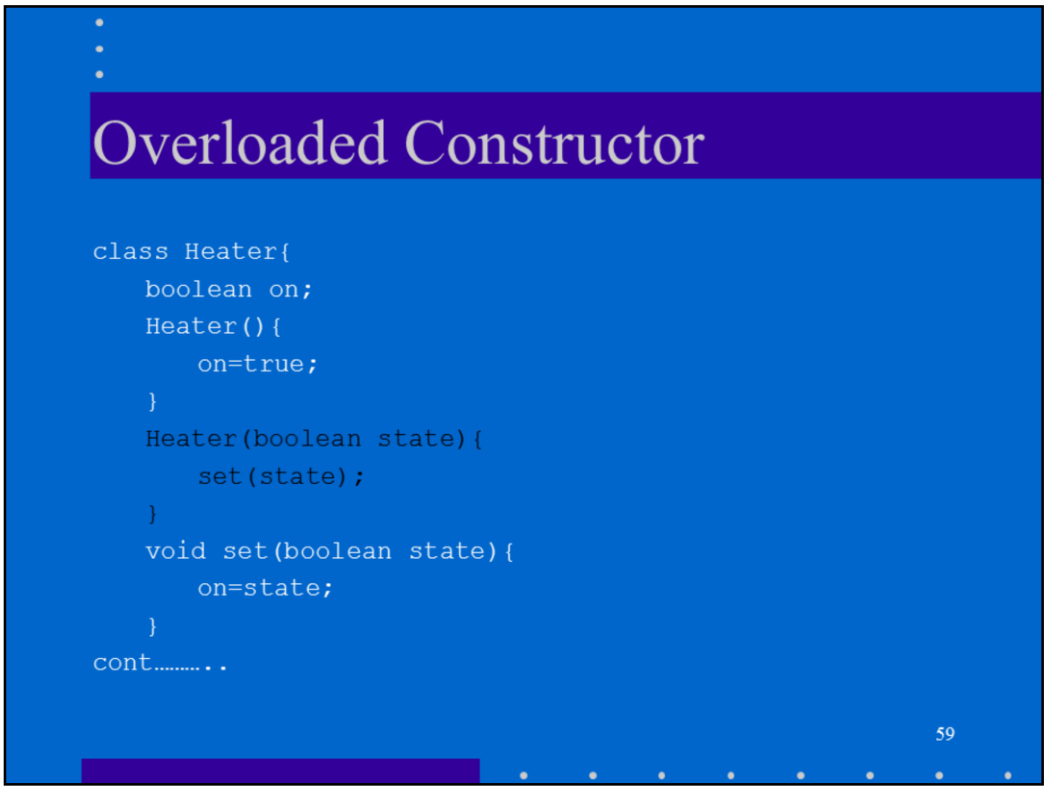
The default constructor created by Java will set all numeric instance variables to 0, all booleans to false and all object references to `null`.



The `Heater` class shown on the slide has a default constructor defined. The instance variable on is no longer set in the declaration but instead is set in the constructor. When an instance of this class is created the constructor will set the value of `on` to `true`.

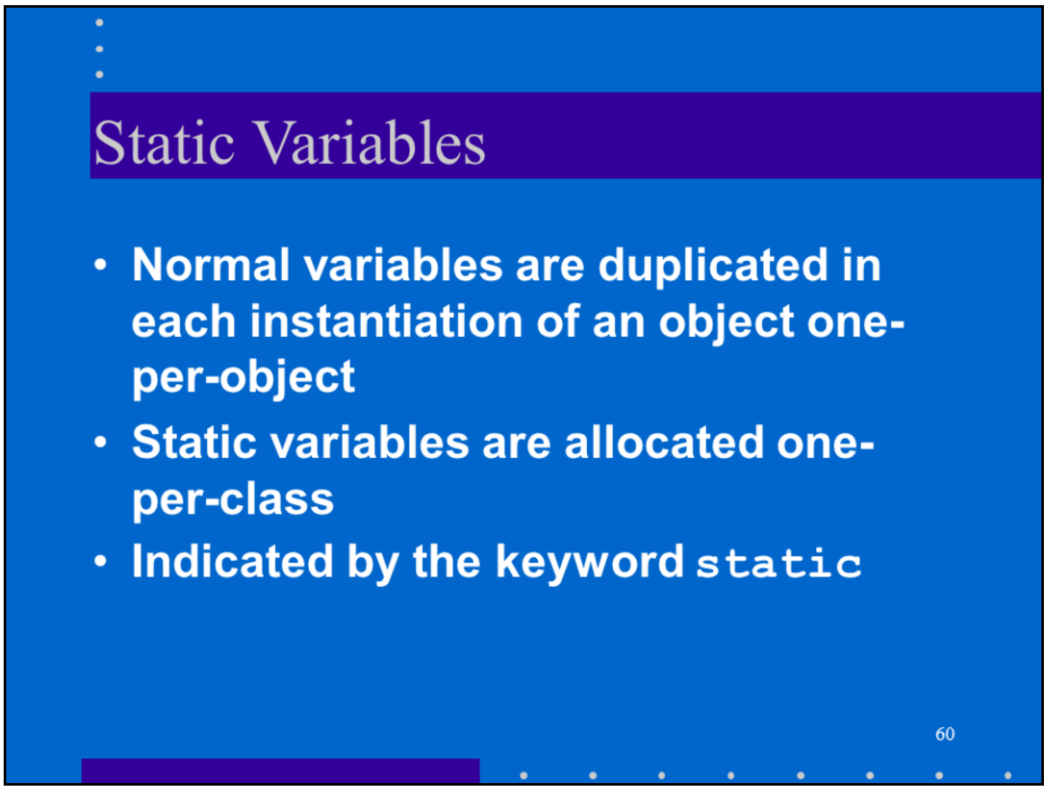
In the previous definition of `Heater` the value of `on` was always initially `true`. This is also the case for the code shown on the slide. The constructor could be rewritten, however, to vary the initial value of `on` depending on the time of day the code is executed. Constructors are more flexible than default assignments.

Note that the constructor does not have a return type and has the same name as the class. It has no parameters and is therefore a default constructor.



The code on the slide shows constructor overloading. The default constructor is still available but a second constructor has been added which takes a `boolean` parameter. The second constructor enables the programmer to set `on` to either `true` or `false` when a `Heater` object is created. The constructor selected will depend on whether a parameter is supplied or not.

Again the constructor has the same name as the class and does not specify a return value.

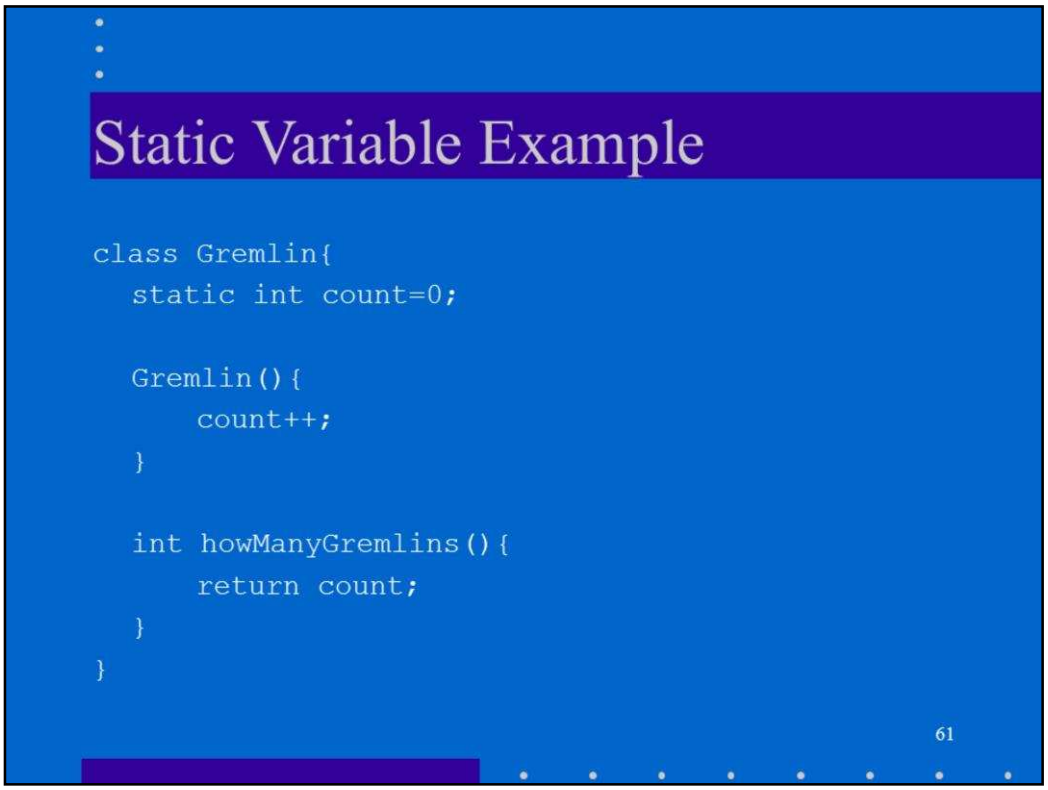

Static Variables
• Normal variables are duplicated in each instantiation of an object one-per-object
• Static variables are allocated one-per-class
• Indicated by the keyword static
60

When an object is created space is allocated for each of its instance variables. Every time a `Heater` is created a new piece of memory is allocated for the `on` variable. If we create 10 `Heaters` then 10 separate `on` variables will be allocated memory. At any time some of these will be set `true` and the others `false`.

It is possible to have a variable which is shared by all the instances of a class. No matter how many instances of the class are created there will be only a single instance of such a variable.

Java and C++ prefix the declaration of this type of variable in the class definition by the prefix `static`.

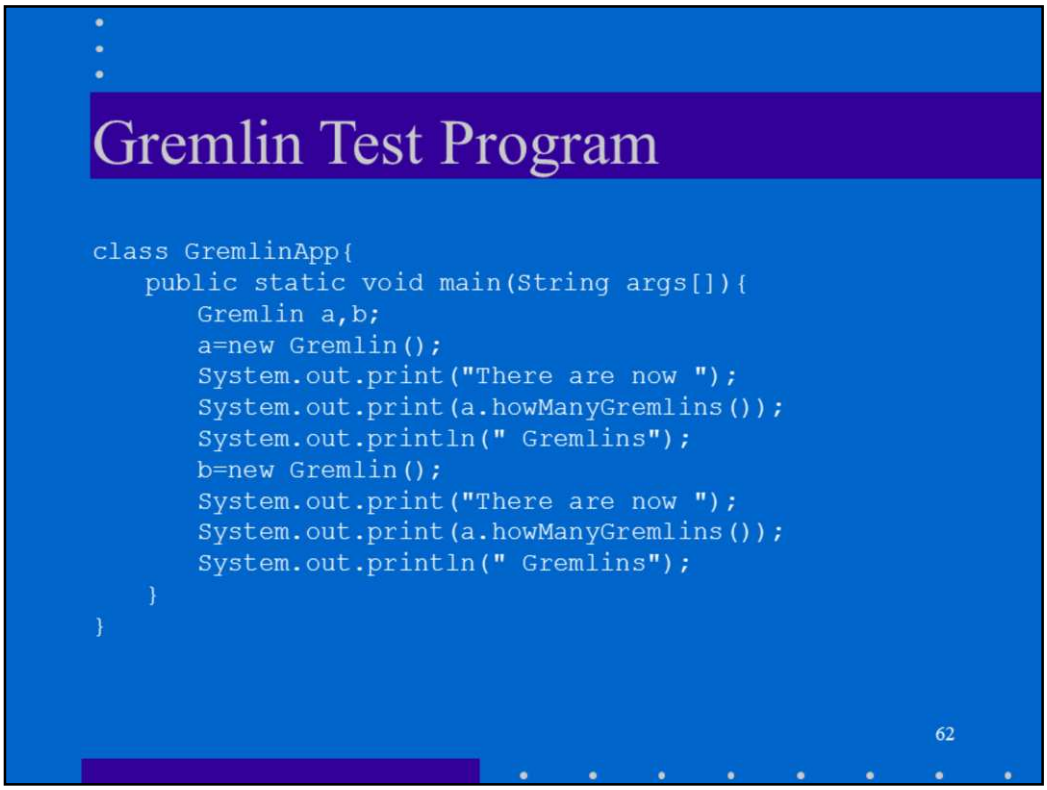
e.g. `static int classVariable;`



This class has a class variable called `count`. No matter how many instances of `Gremlin` are created there will be only one instance of `count`. The constructor for this class increments the `count` variable. Consequently, `count` will always contain the number of `Gremlin`s that have been created.

Each instance of a `Gremlin` will be able to access the `count` variable. The method `howManyGremlins` returns the value of this variable reporting the number of `Gremlin`s that have been created.

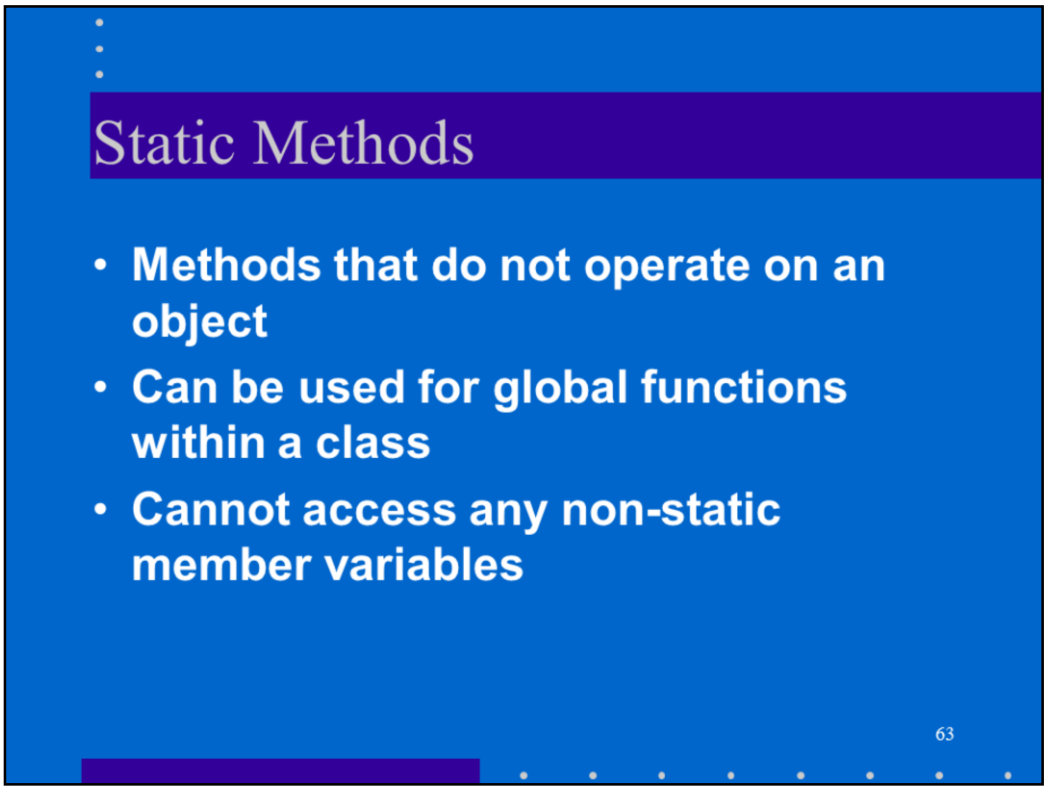
If an additional constructor were to be added to the `Gremlin` class the programmer would have to remember to increment `count` in the new constructor if `count` was to be kept accurate.



This program shows how the `Gremlin` class can be used within an application.

A new `Gremlin` is created (this will be explained later). The new object is asked how many `Gremlin`s have been created. The answer will be 1. A second `Gremlin` is created. The original object is again asked how many `Gremlin`s have been created, this time the answer will be two.

The answer is correct because both `Gremlin` objects have access to the single counter.

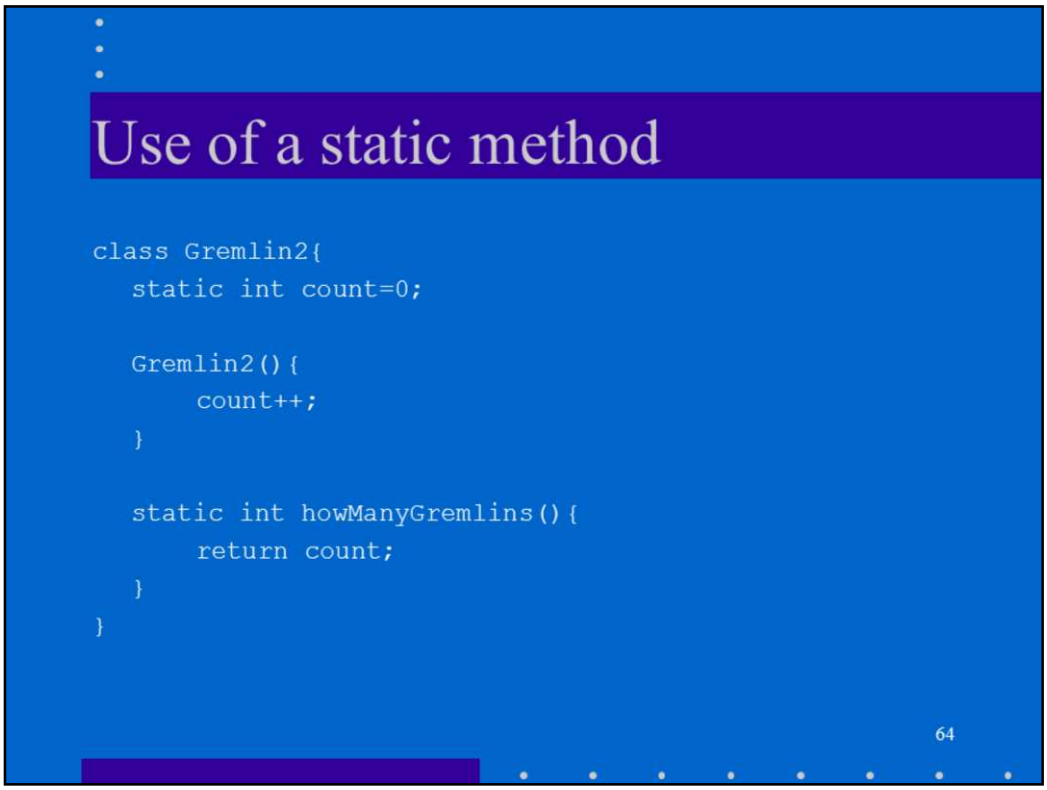

Static Methods
• Methods that do not operate on an object
• Can be used for global functions within a class
• Cannot access any non-static member variables
63

There are some methods which do not operate on data contained within an object. For example methods that convert metres to feet or litres to pints should not be linked to any particular object as the same formulae apply equally to a range of objects.

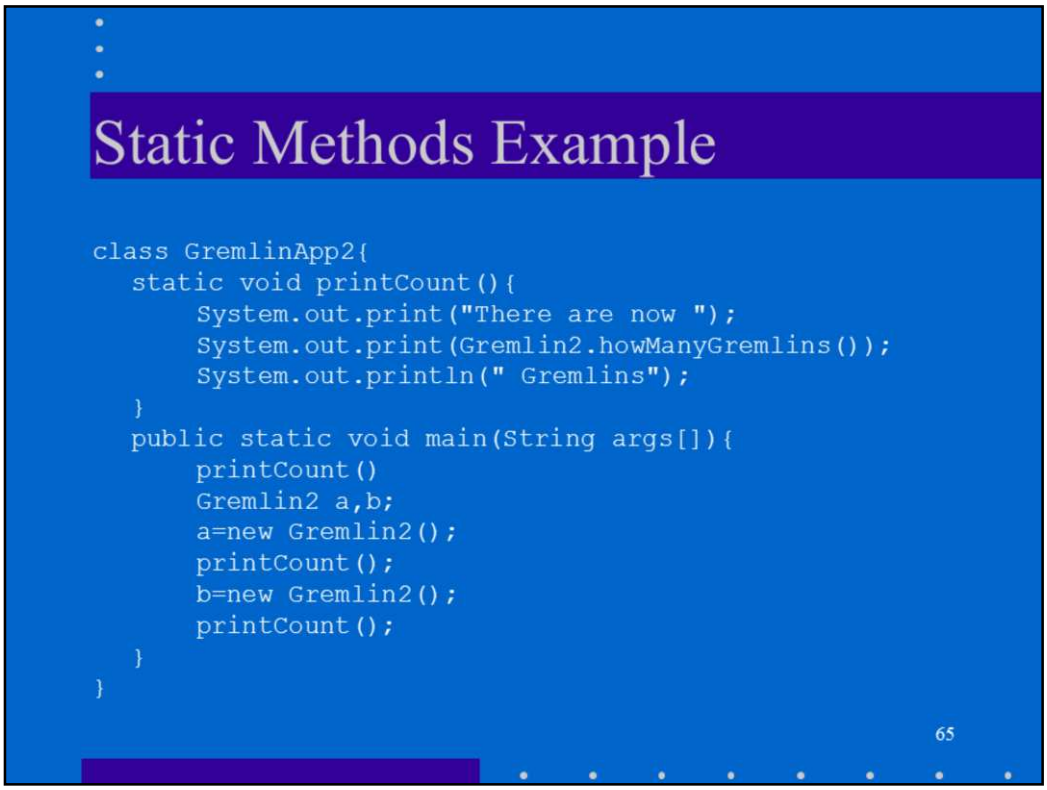
Methods which do not need to operate on the data within a class can be declared as `static`. They can be used within the class they are declared in as global functions or else used by other classes.

`static` methods can only access class variables which have also been declared as `static`. It is not necessary to create an instance of a class to use its `static` methods.

```
class Convert{
      static float centimetresToMetres (int centimetres){
            return centimetres/100.0f;
      }
}
class ConversionTest{
      public static void main(String args[]){
            System.out.print("45 centimetres is equal to ");
            System.out.print(Convert.centimetresToMetres(45));
            System.out.println(" metres");
      }
}
```



In the Gremlin class, `howManyGremlins()` should be declared as `static`. This is because the only variable `howManyGremlins()` accesses is `count` which is a `static` variable. The method does not access any instance variables.



In `GremlinApp` the code to print out the current number of `Gremlin`s is repeated inline. None of this code accesses any instance variables of the `GremlinApp` class. It can therefore be placed in a `static` method. Ordinary method invocations need to be applied to an object of the class (as in *`object.methodName()`* where *`object`* is an object reference). `static` methods are simply invoked by name as in `printCount()`.

If `printCount` was used by another class other than Gremlin2, then it would need to be be qualified by a class name as in `Gremlin2.howmanyGremlins()`.

The example uses class `Gremlin2` in which `howManyGremlins()` is declared as `static`. Given this declaration it is possible to print out the number of `Gremlin`s even before any have been created (the answer will be zero!).

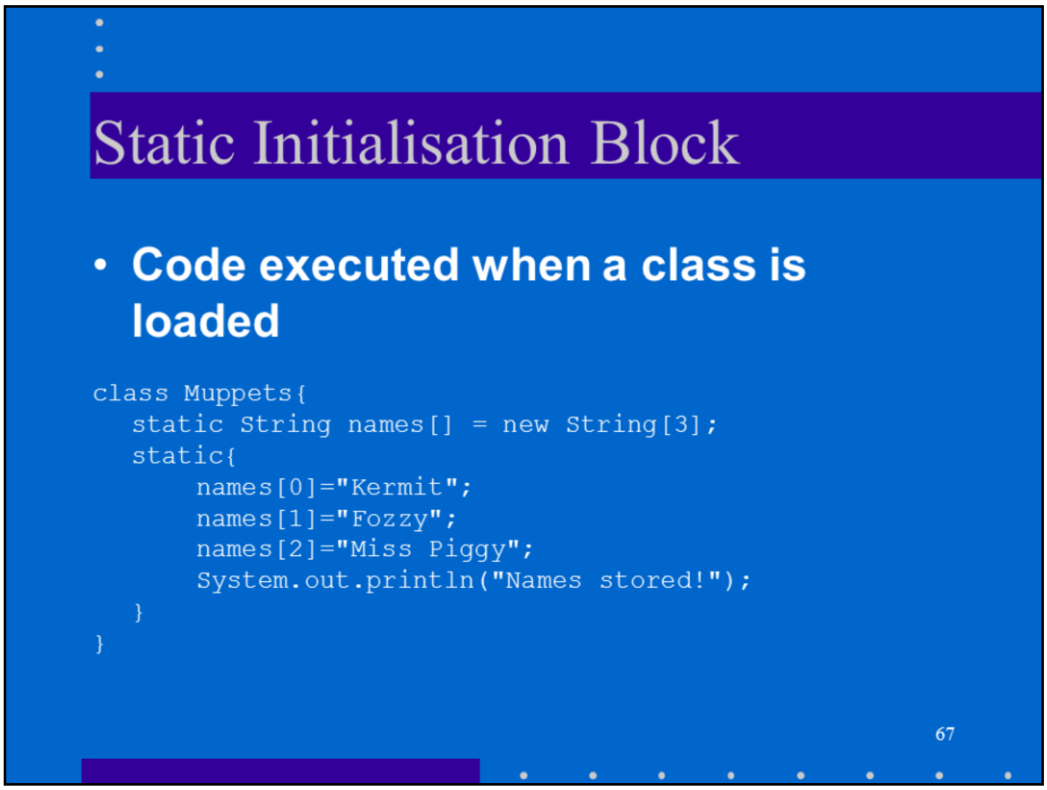
Initialisation Block

- **Intialisation code**

```
class Heater{
  boolean on;
  {
        on=true;
        System.out.println("Heater is on");
  }
  boolean get(){
        return on;
  }
}
```

66

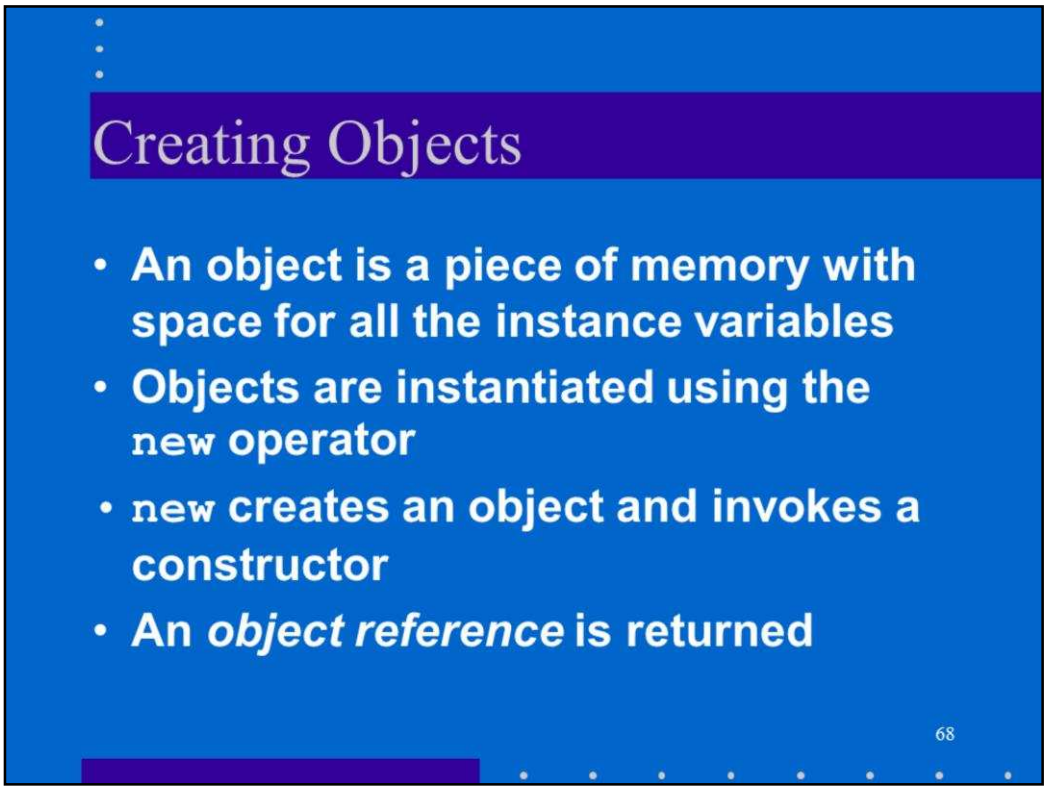
Here a block of code delimited by { and } has been added to the class. This coded is executed every time an object is created (normally many times). Such a block could be used to hold code which is common to every constructor for the class



Java classes are loaded dynamically, that is at the time which they are used. If the flow of control through a program avoids the lines of code that use class X then class X will not be loaded.

Code located within a class which is declared as `static` is executed when the class is loaded (normally once only). In the example shown here the code initialises an array and then prints a message out.

This feature of Java is most commonly used for initialising static arrays.


Creating Objects
• An object is a piece of memory with space for all the instance variables
• Objects are instantiated using the new operator
• new creates an object and invokes a constructor
• An object reference is returned
68

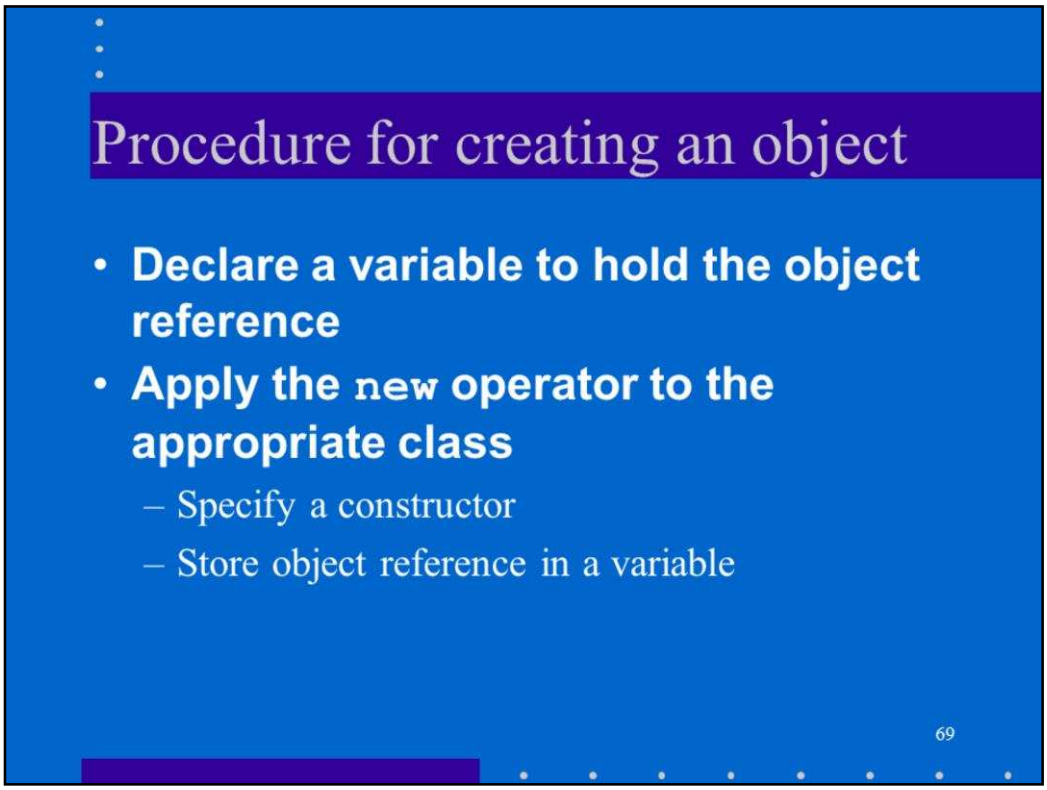
Class definitions define the pattern an object will have when it is created. In particular, they state the storage requirements of the object.

Object creation is the point at which memory is actually allocated to the object. Every object must be created before it can be used.

Objects are created by applying the `new` operator to a class name.

At the same time the programmer specifies which constructor should be used to create the object. If no constructors have been defined for the class then this must be the default constructor.

Java allocates space for the object and then returns a reference to the object.


Procedure for creating an object
• Declare a variable to hold the object reference
• Apply the new operator to the appropriate class
– Specify a constructor
– Store object reference in a variable
69

In order to create an object the programmer must first declare a variable to hold the object reference that Java will return after the object's creation. This must be of the correct type.

Once declared, class names can be used in the same way as basic types. So just as it is possible to declare `x` as

```
int x;
```

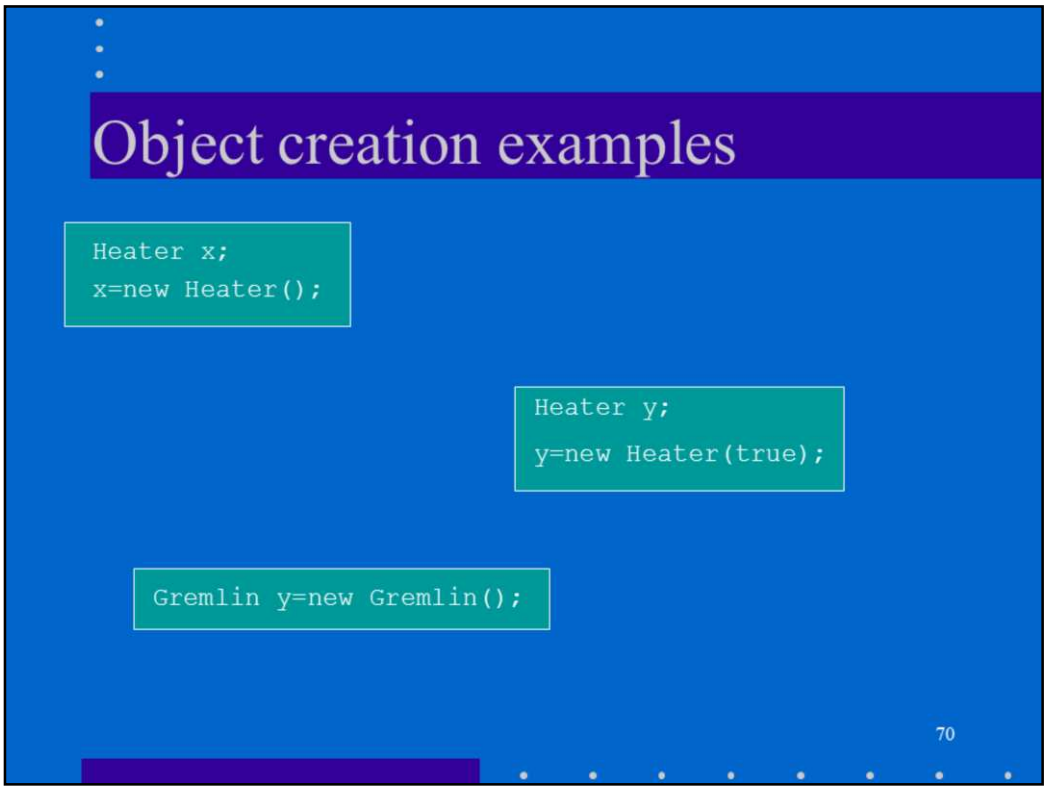
it is possible to declare y as

```
Heater y;
```

In this case `y` is not an instance of a `Heater` but a variable which may contain a reference to an object of type `Heater`.

The programmer then specifies a constructor to be used by the `new` operator, this will of course have the same name as the class. The parameters supplied will determine which constructor is used.

The result of applying the `new` operator to the class is stored in the variable declared in the first step.



There are three object creations shown on this slide.

```
Heater x;
x=new Heater();
```

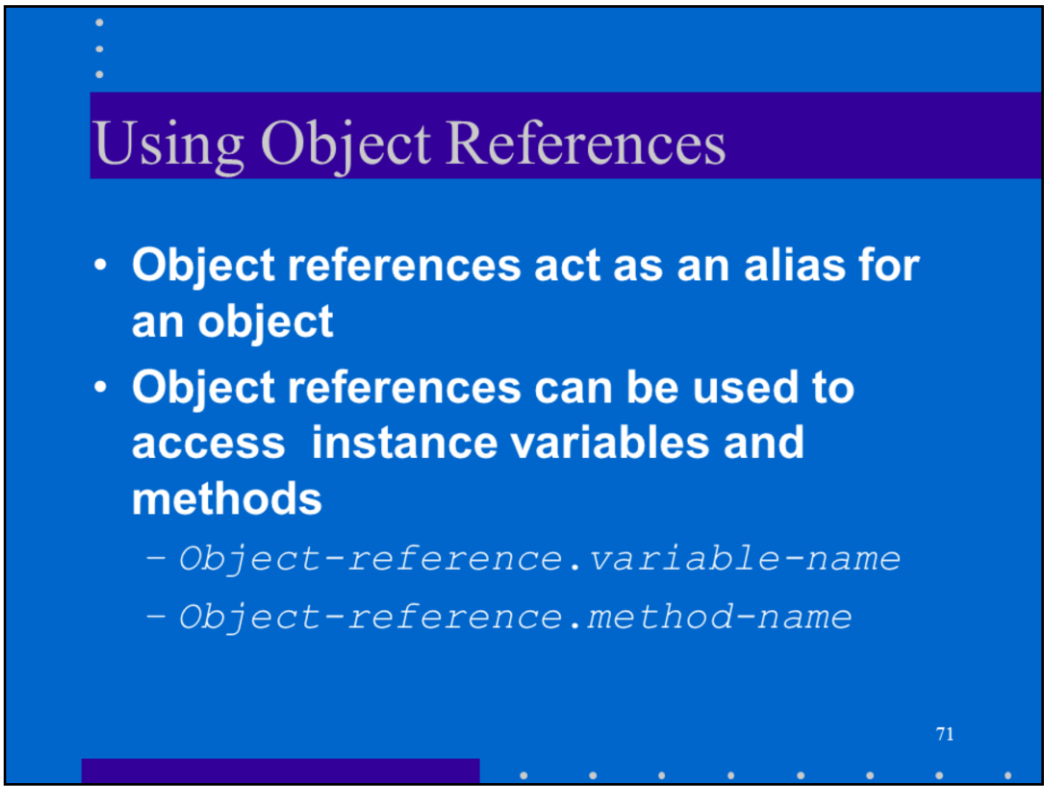
Here `x` is declared as an object reference to an object of type `Heater`. The object is created using the `new` operator and the default constructor and its object reference is stored in `x`;

```
Heater y;
y=new Heater(true);
```

This code is similar but in this case the other constructor is used. Java recognises this from the fact that a `boolean` parameter has been supplied.

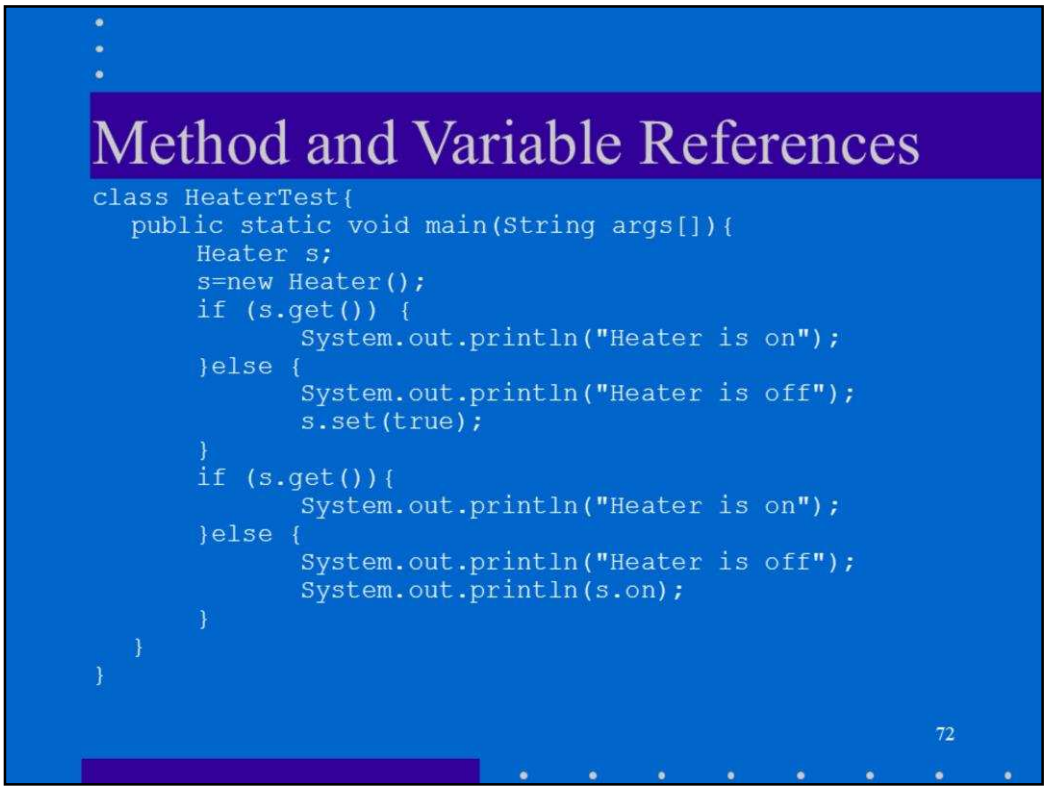
```
Gremlin y=new Gremlin();
```

A `Gremlin` is created using the default constructor. Here the entire operation is completed in a single line of code.


Using Object References
• Object references act as an alias for an object
• Object references can be used to access instance variables and methods
– Object-reference.variable-name
– Object-reference.method-name
71

Once created an object reference is a means of accessing and manipulating the object. It behaves a little bit like a pointer in C++, however it doesn't have to be dereferenced and it can't be manipulated arithmetically.

Object references are used to access the methods and variables of the object they represent.

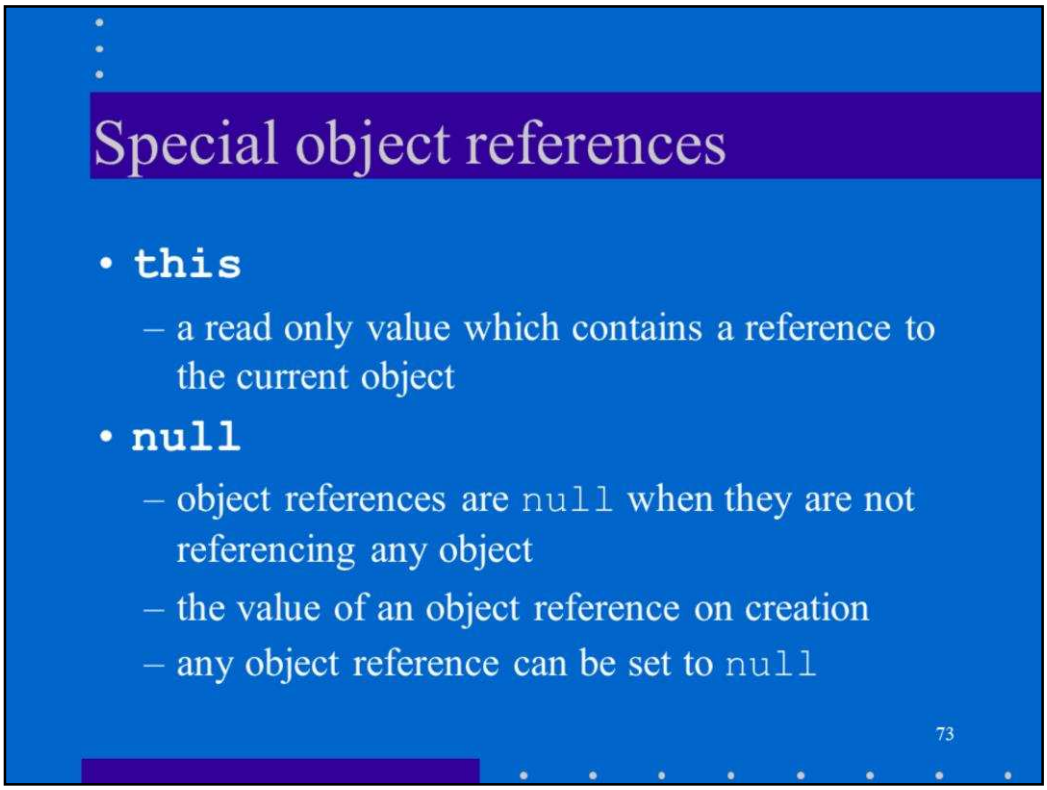


This code creates a `Heater` object and stores a reference to it in the variable `s`.

The value of `s.get()` is then tested. The `get` method of `Heater` returns the value of the `boolean` instance variable `on`. In this case the value of `on` in the object referenced by `s` is returned. If this is `true` then the program reports that the heater is on, otherwise it will report the heater is off. Since `on` is initialised to `true` we would expect the program to report that the heater is on.

The `set` method of `Heater` is now applied to the object referenced by `s`. This sets the object's `on` variable to `true` (no change). Again the `get` method is used to test the value of the `on` variable in the object referenced by `s`. Again we would expect the heater to be on.

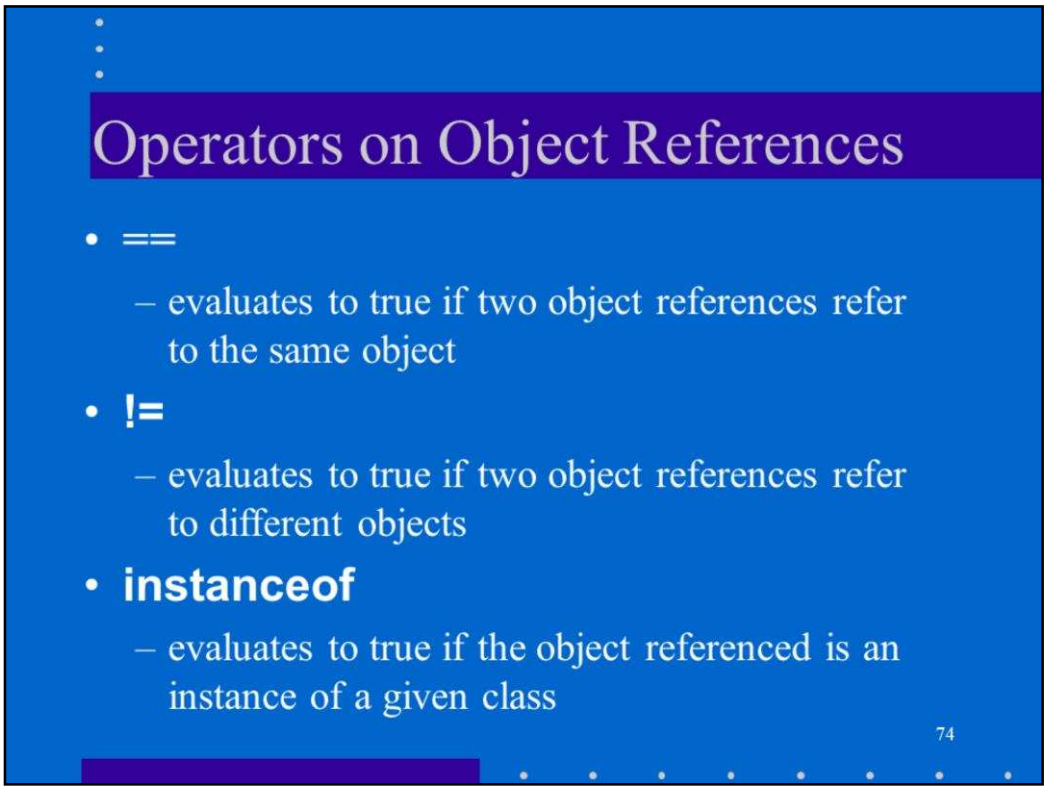
Finally the program prints out the value of the `on` instance variable in the object referenced by `s`. There could be dozens of `Heater` objects in existence and therefore dozens of `on` instance variables but only the version referenced by `s` is printed.


Special object references
• this
– a read only value which contains a reference to the current object
• null
– object references are null when they are not referencing any object
– the value of an object reference on creation
– any object reference can be set to null
73

There are two pre-defined object references.

`this` when used within a method refers to the current object. For example if a method invoked as `x.method()` uses `this` then `this` refers to the same object as `x`.

`null` is a value which when stored in a variable indicates that the variable is not referencing any object.


Operators on Object References
• ==
– evaluates to true if two object references refer to the same object
• !=
– evaluates to true if two object references refer to different objects
• instanceof
– evaluates to true if the object referenced is an instance of a given class
74

Just as two pointers may point at the same memory location, so two object references may refer to the same object.

If `x == y` is `true` then `x` and `y` refer to the same object. If `x` and `y` refer to different objects of the same class which have instance variables that are identical then `x == y` is `false` because the objects involved are different objects.

`!=` tests for object references that are not equal i.e. it is `true` when they refer to different objects.

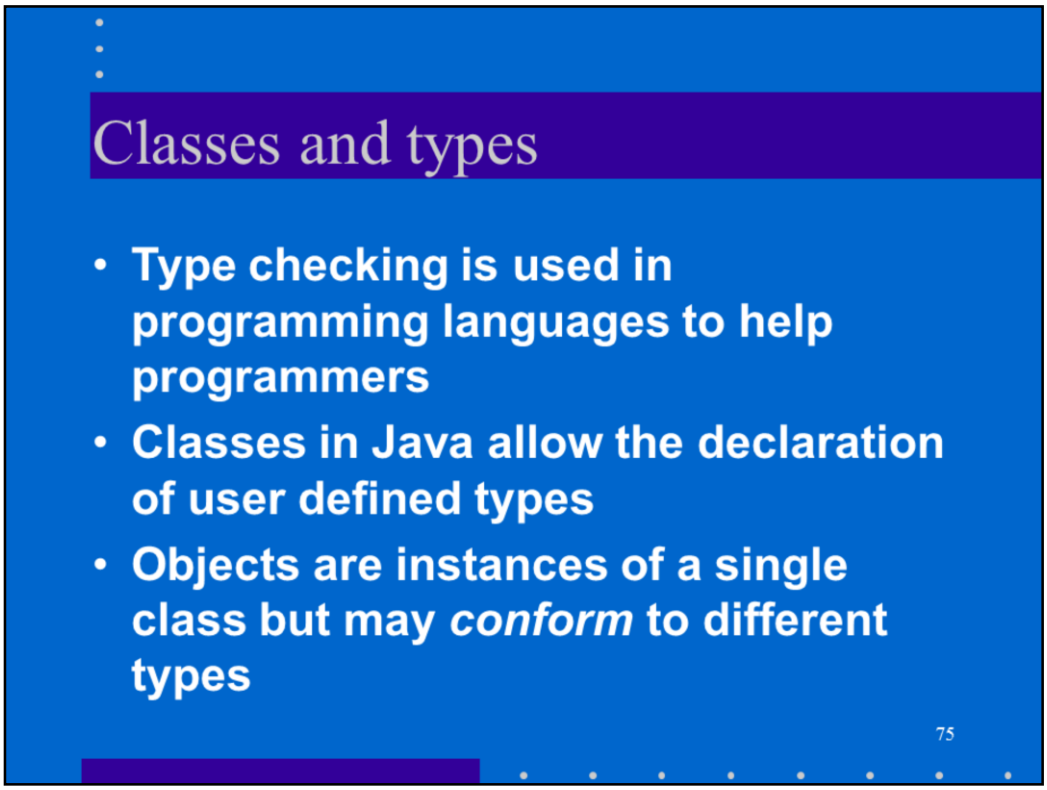
If we execute the following code:

```
Gremlin x = new Gremlin();
```

then

`x instanceof Gremlin` will evaluate to `true` because `x` references an object that is an instance of the `Gremlin` class.

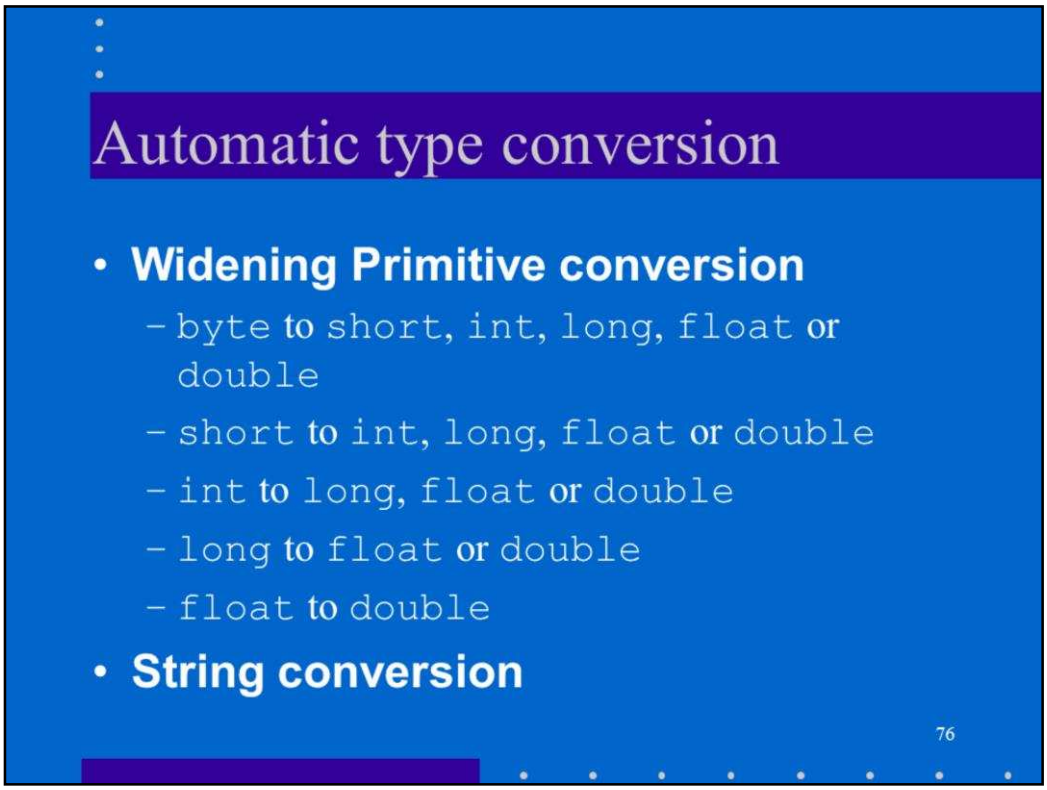
Note that use of inheritance (see next lecture) makes `instanceof` a little more complicated!


Classes and types
• Type checking is used in programming languages to help programmers
• Classes in Java allow the declaration of user defined types
• Objects are instances of a single class but may conform to different types
75

In C++ every variable has a type (`int`, `float` etc.). The same is true of Java. We often say that it doesn't make sense to compare apples with pears. The type mechanism is used by the compiler to check that a programmer isn't trying to a carry out a nonsensical operation. For example, comparing a number with a character is not normally a useful operation. Initially, when you are learning programming type errors are annoying because you know what you wanted to do and the compiler seems to be too dumb to understand you. Type checking can, however, prevent programmers from making quite major errors.

Whenever you declare a class in Java you are in effect also creating a user defined type i.e. adding to the types the Java compiler knows about. In the same way that the compiler ensures that you make the correct use of basic types it will check that you make correct use of any user defined types.

We shall see later (next lecture), however, that classes and types are not quite the same thing. An object is created from a single class but may *conform* to several types. An object may be used wherever one of the types it conforms to is permitted. In this case the type expected and the class of the object may be different.


Automatic type conversion
• Widening Primitive conversion
– byte to short, int, long, float or double
– short to int, long, float or double
– int to long, float or double
– long to float or double
– float to double
• String conversion
76

Given the expression:

```
x + y + z
```

A very strict interpretation of typing would insist that `x`, `y` and `z` all have the same type. Java, however, will allow automatic type conversion in this case. Automatic type conversion is allowed between numeric built in type a and numeric built in type b if type b is capable of holding a greater range of numbers than type b. This encompasses all the possible conversions shown on the slide.

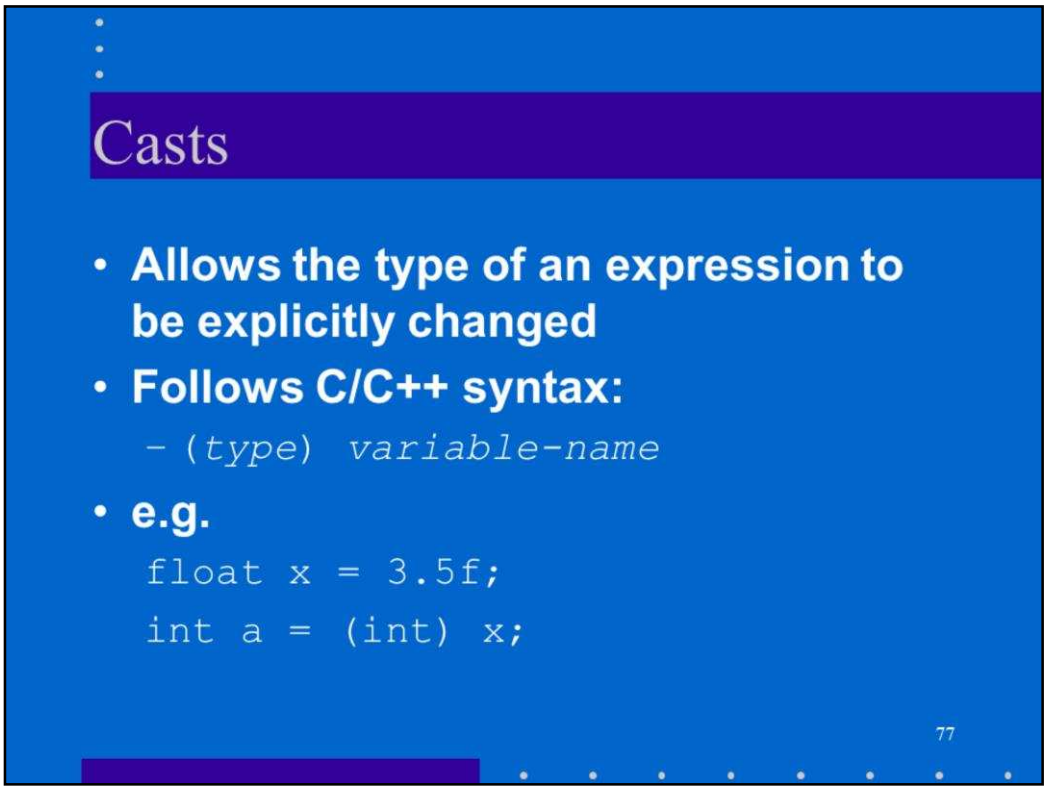
In the expression above, if `x` has type `int`, `y` has type `long` and `z` has type `float`, then `x` and `y` will be automatically converted to type `float` and the result of the expression will be of type `float`.

In expressions which contain a '+' operator where one of the operands is a `String`, the other operand will be automatically converted to a `String` and the result of the expression will be a `String`.

The result of the expression:

```
"Hello" + 1
```

will be the String `"Hello1"`.



The automatic type conversions are those which are always safe and will always work. There are occasions, however when a programmer may know exactly what type conversions are required and may want to request the compiler to carry out the type conversion. This is done via a mechanism known as '*casting*'. The syntax comes via C++ from C. The name of the desired type is placed in brackets in front of the expression to be converted.

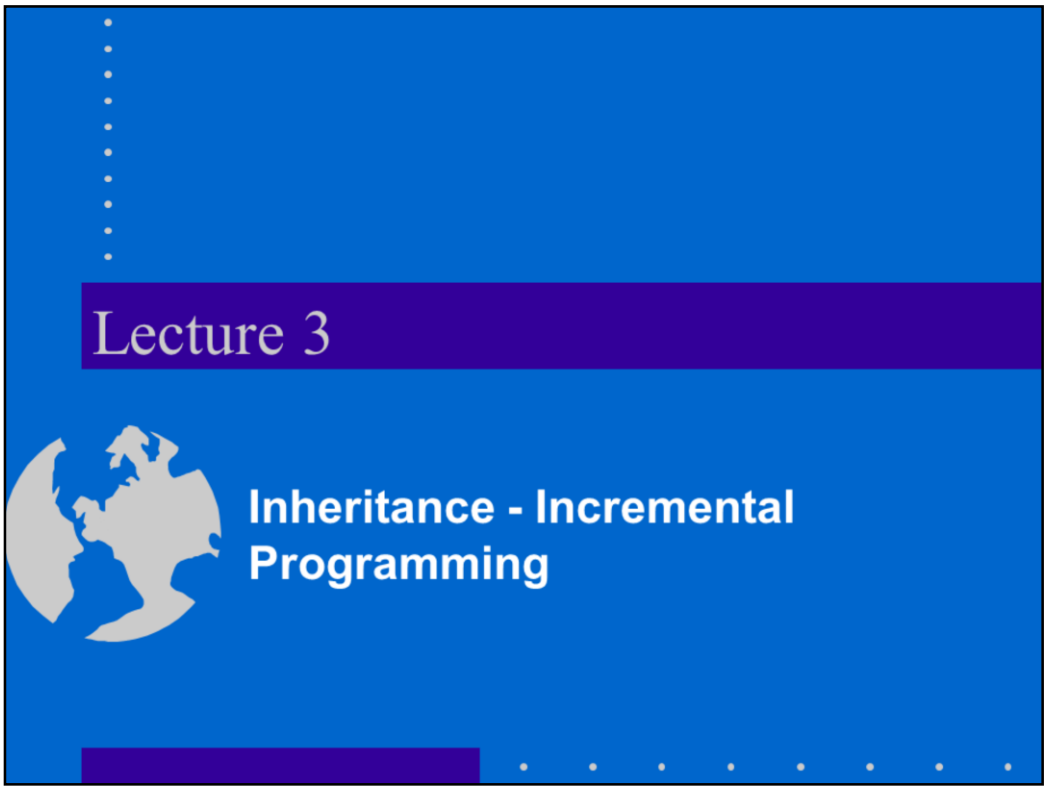
Type names permitted include both simple and derived types.

Programmers must know what they are doing when they use a cast. For example, in the code shown on the slide the result of casting the floating point value 3.5 to an `int` is 3, i.e. the figures after the decimal point are thrown away (they are not rounded).

Certain casts are permitted, other casts are not. For example, an attempt to cast a `String` to an `int` will not be allowed. The rules for casting will be dealt with at the appropriate points in the lectures. Some illegal casts will be flagged by the compiler, others can only be detected at run time.

Summary

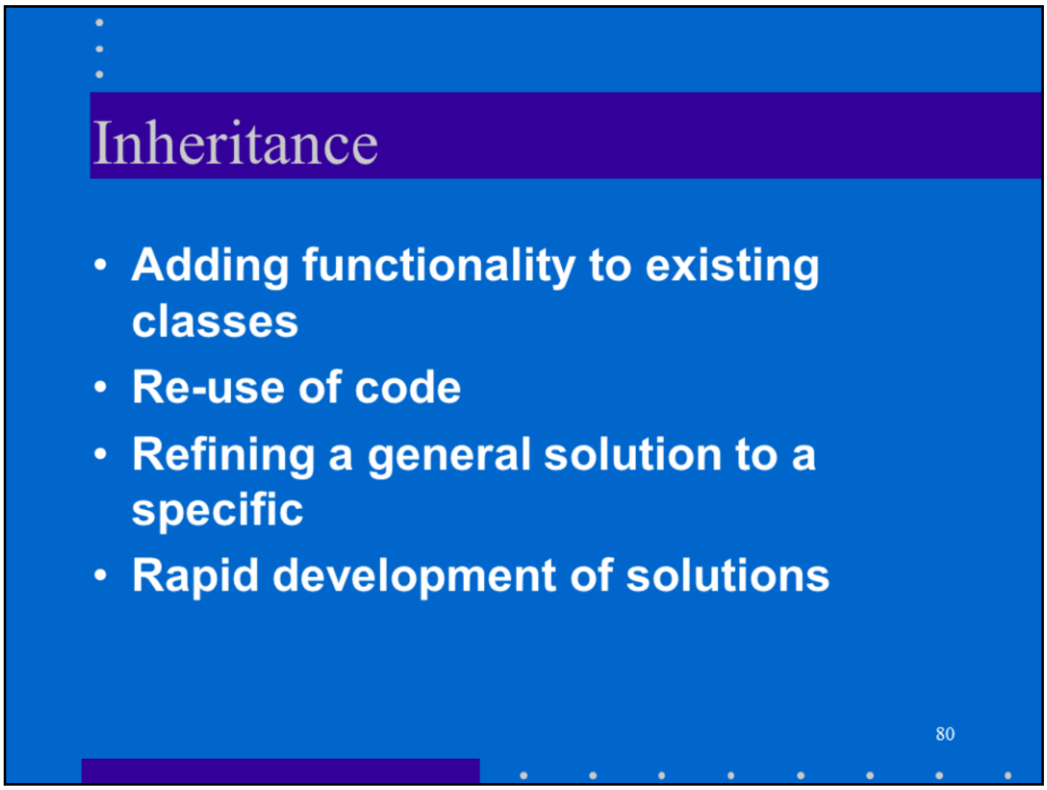
- Objects and Classes
- Class definition
- Methods
- Method Overloading
- Instance variables
- Constructors
- `final` variables
- `static` methods
- `static` variables
- Object creation
- Object references



The standard Java Development Kit provides a rich variety of classes which can be used to create applets and applications. It is unlikely, however, that exactly the classes you want to use will be available. Inheritance provides a mechanism whereby classes can be adapted to meet the needs of a programmer. In addition, inheritance provides an effective way by which existing code can be reused.

The use of inheritance encourages a style of programming where a programmer rather than developing each program from scratch, simply adds to and adapts existing software. In addition, programmers can create their own classes and then define new classes based upon them.

In Inheritance the original source code is unaltered but additional functionality is provided in a class which references the original code.


Inheritance
• Adding functionality to existing classes
• Re-use of code
• Refining a general solution to a specific
• Rapid development of solutions
80

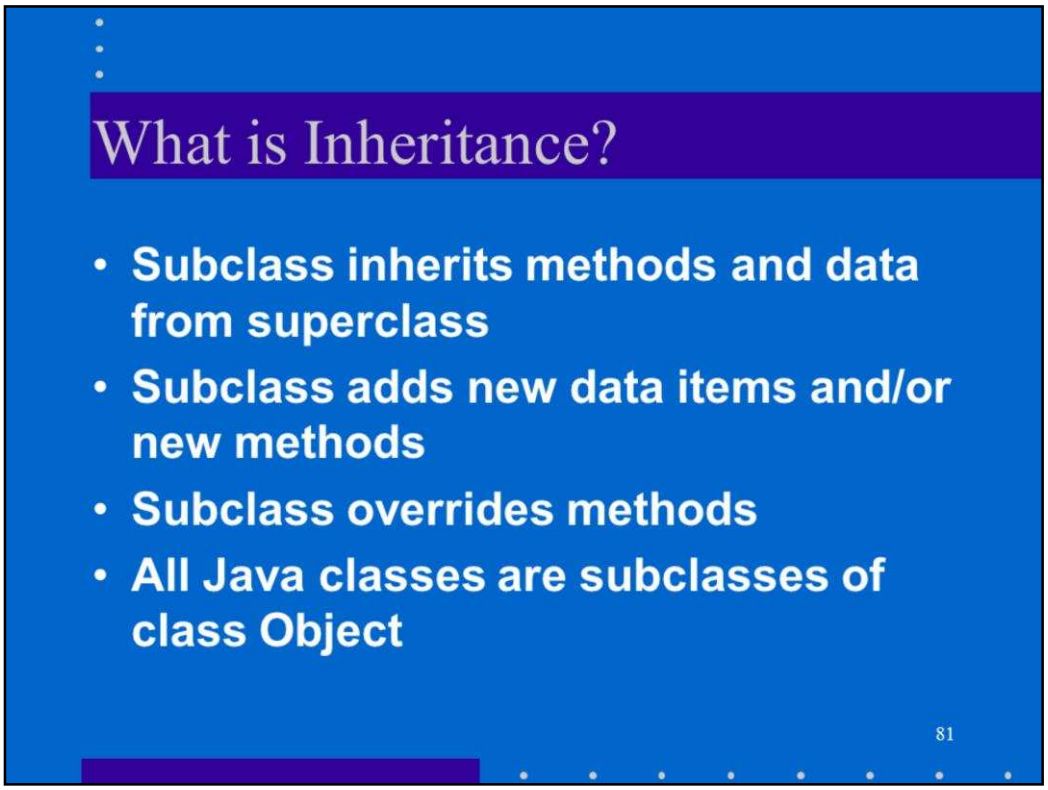
Inheritance has a number of purposes and benefits.

It enables a programmer to develop new functionality without forcing a rewrite of code that already exists. It encourages a development style which enhances software already developed by other programmers to meet the needs of a new application area.

The very nature of inheritance is that it makes the maximum possible use of generic code in order to underpin application specific code. This means that code is often re-used. The more code that is re-used, the less a project costs.

Inheritance implies a method of development where a general approach is tuned to a particular set of circumstances. This contrasts with a package based approach where the application area is made to fit in with what the software does! OO solutions can be easily tuned to match special requirements.

Inheritance allows a programmer to make use of fully tested software components. Consequently, applications require much less new code, this means that they can be developed much faster.

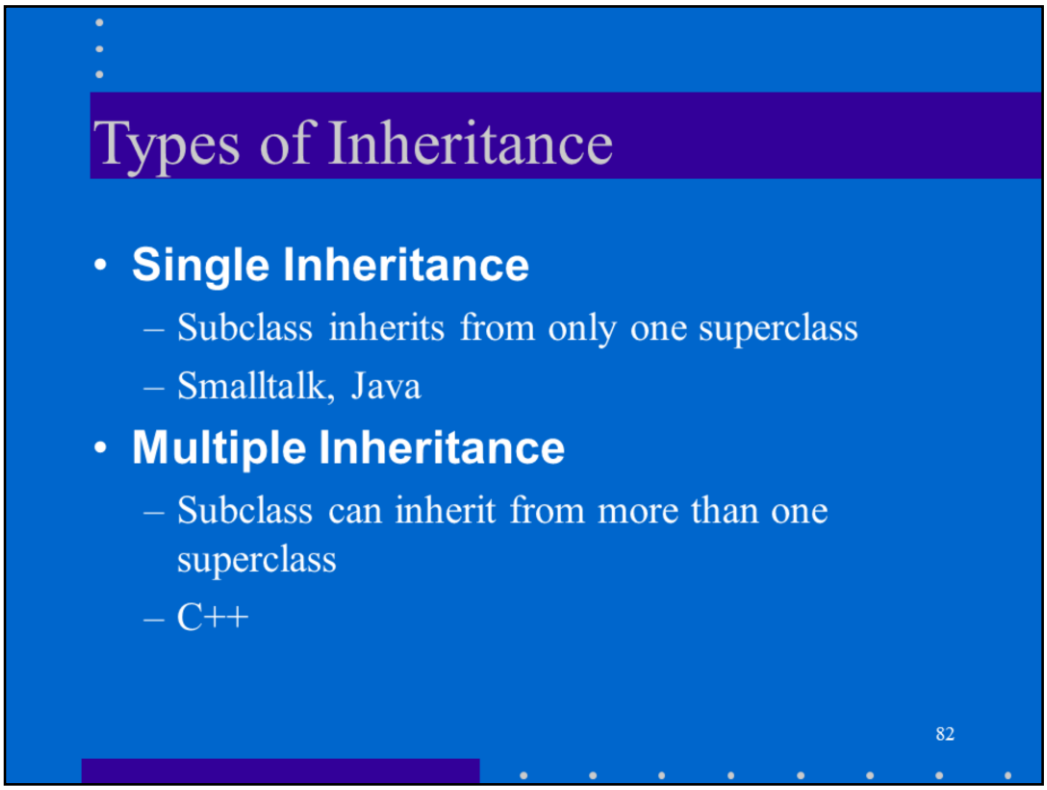

What is Inheritance?
• Subclass inherits methods and data from superclass
• Subclass adds new data items and/or new methods
• Subclass overrides methods
• All Java classes are subclasses of class Object
81

Inheritance is a mechanism whereby a new class is created as an extension of an existing class. The original class is called a superclass, the new class is called a subclass. Usually a subclass is a specialisation of its superclass, for example, class Car will be a specialisation of its superclass Vehicle.

Without any further declarations, the new class has access to the methods and data of the class it inherits from(extends). It may then add further data items and further methods. Almost certainly where a class adds new data items it will also add new methods to process the new data variables.

A subclass may also supply its own version of a superclasses method by *overriding* the method.

Inheritance is central to Java. All classes in Java inherit from a class called `Object` (even though this is not declared explicitly). This means that all objects in Java understand messages which correspond to the methods of `Object`.



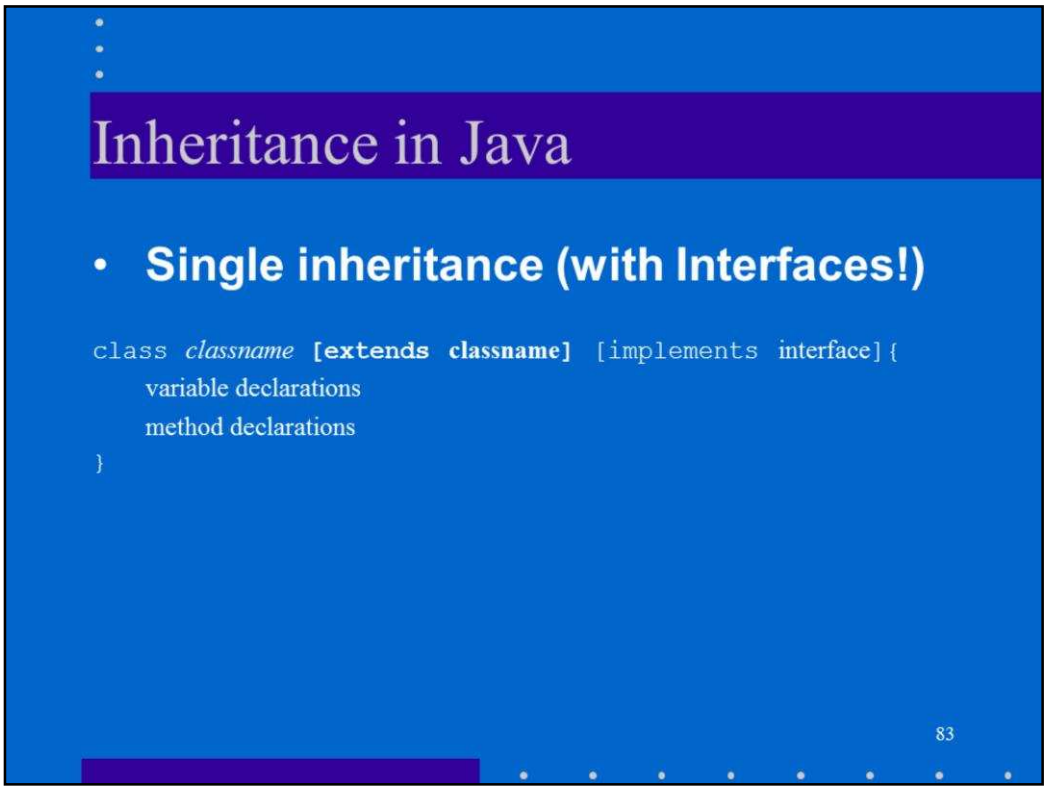
There are two types of inheritance found in object-oriented languages.

Single inheritance where a subclass inherits from one subclass only. This was the original form of inheritance and is found in languages such as Smalltalk and VB.Net.

In multiple inheritance a subclass can inherit from a number of superclasses. C++ permits multiple inheritance.

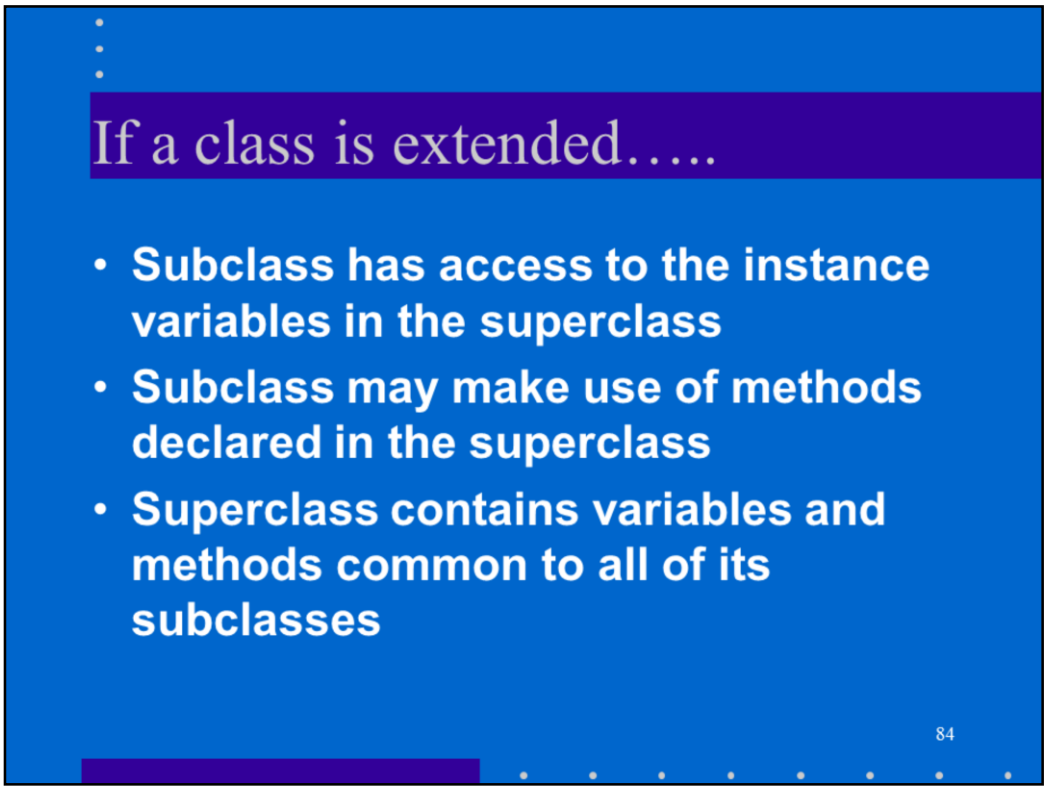
Multiple inheritance is generally considered to be the most powerful form of inheritance but it is extremely difficult for programmers to handle.

Bjarne Stroustrup the creator of C++ says the major motivation for adding multiple inheritance to the language was that it was relatively easy to do and nobody thought he could do it! ("The Design and Evolution of C++", B. Stroustrup, Addison-Wesley, 1994, ISBN: 0-201-54330-3). Apparently the issue of whether it was useful was not a motivation!



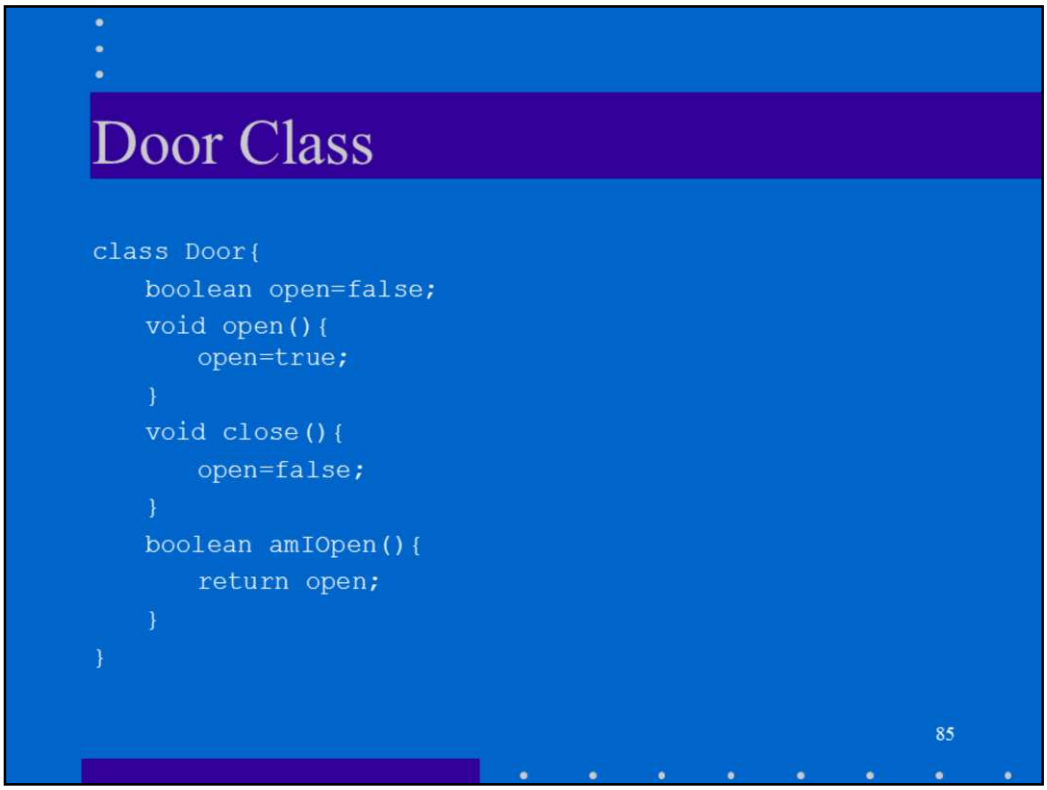
The creators of Java decided that in the interests of simplicity they would only include single inheritance in the language. The Java idea of interfaces, however goes some way to providing some of the functionality that multiple inheritance provides.

The syntax for inheritance is the `extends` phrase of the class definition. The class being declared becomes the subclass of the class it *extends* (the superclass). A class may only extend one other class (single inheritance).


If a class is extended…..
• Subclass has access to the instance variables in the superclass
• Subclass may make use of methods declared in the superclass
• Superclass contains variables and methods common to all of its subclasses
84

In general a subclass will have access to the instance variables of its superclass (but see the section on encapsulation later in this lecture). Likewise the subclass will automatically support all the methods of the superclass (subject to any encapsulation restrictions).

The idea of designing for inheritance is to identify a set of classes who share common methods and data. These common items can then be placed into a superclass. The code in the superclass will then be re-used by all the subclasses.

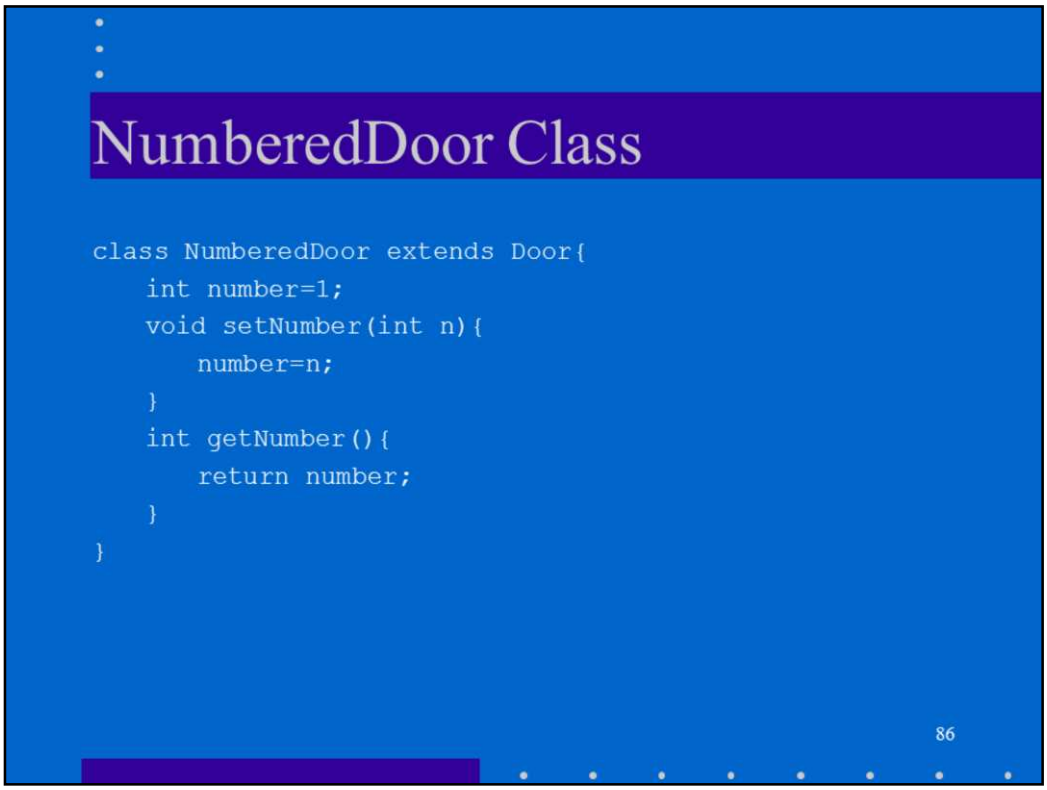


Here we have a class called `Door`. The only thing of interest in a door is whether it is open or closed. The `boolean` variable open records the status of the door. If `open` is `true` then the door is open else it is closed.

The method `open()` sets the value of the variable `open` to `true` and opens the door.

The method `close()` sets the value of the variable `open` to `false` and shuts the door.

The method `amIOpen()` returns the status of the door.

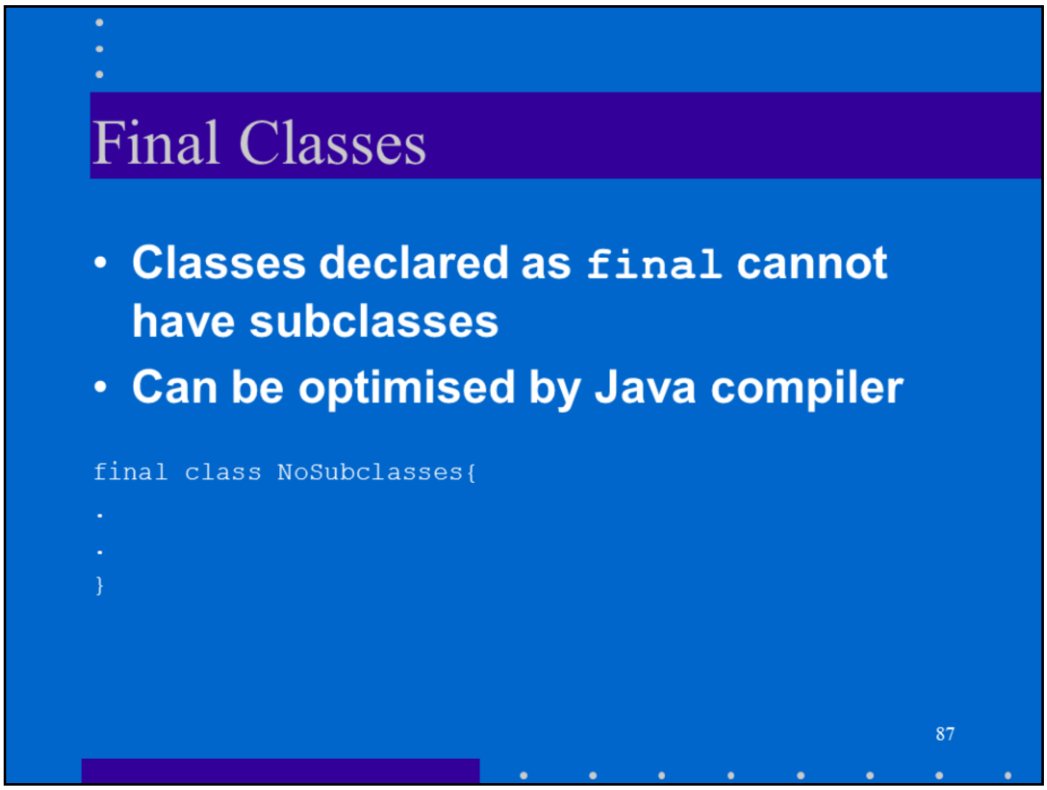


In a `NumberedDoor` we are interested in whether the door is open or shut and also what number it has.

It would be possible to write a class `NumberedDoor` which contains the declarations of the instance variable `open` and the methods of `Door`. The purpose of inheritance is to make that unnecessary. `NumberedDoor` inherits (via the `extends` keyword) from `Door`. So `NumberedDoor` has an instance variable `open` and understands all the methods of `Door`.

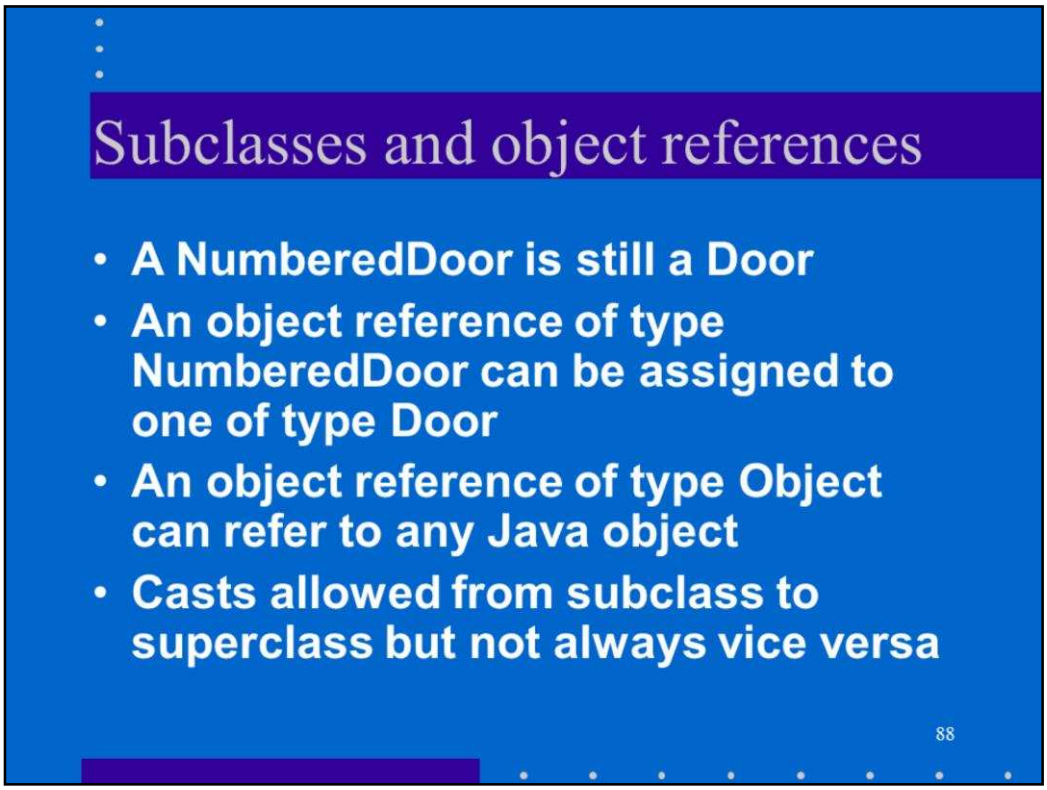
It also adds an instance variable `number` and two methods `setNumber()` and `getNumber()`.

`NumberedDoor` is an enhanced version of `Door`.



For a variety of reasons (a mixture of security concerns and commonsense) there may be classes that should never become superclasses. If a class definition is prefixed by the keyword `final` then it cannot have subclasses. An attempt to extend it will be flagged as a compiler error.

The compiler can optimise class definitions that have been labelled `final`.


Subclasses and object references
• A NumberedDoor is still a Door
• An object reference of type NumberedDoor can be assigned to one of type Door
• An object reference of type Object can refer to any Java object
• Casts allowed from subclass to superclass but not always vice versa
88

When inheritance is used, an object may be deemed to have a number of types. An object of a subclass is regarded as being of the same type as its superclass. So a `NumberedDoor` can be treated as a `Door`. The converse is not true, a `Door` is not a `NumberedDoor` because `NumberedDoor` supports more methods than `Door`.

In lecture two variables were used to reference objects of appropriate types. An object reference of type `Door` could also refer to a `NumberedDoor`. An object reference of type `NumberedDoor` could not refer to a `Door`.

Since every Java object inherits from class `Object`, an object reference of type `Object` can be used to refer to any Java object.

Any object in Java belongs to the class used to create it. The object created in the statement:

Door d = new NumberedDoor();

is a NumberedDoor which is being referred to by an object reference variable which may reference Door objects. Since NumberedDoor is a door this is permitted.

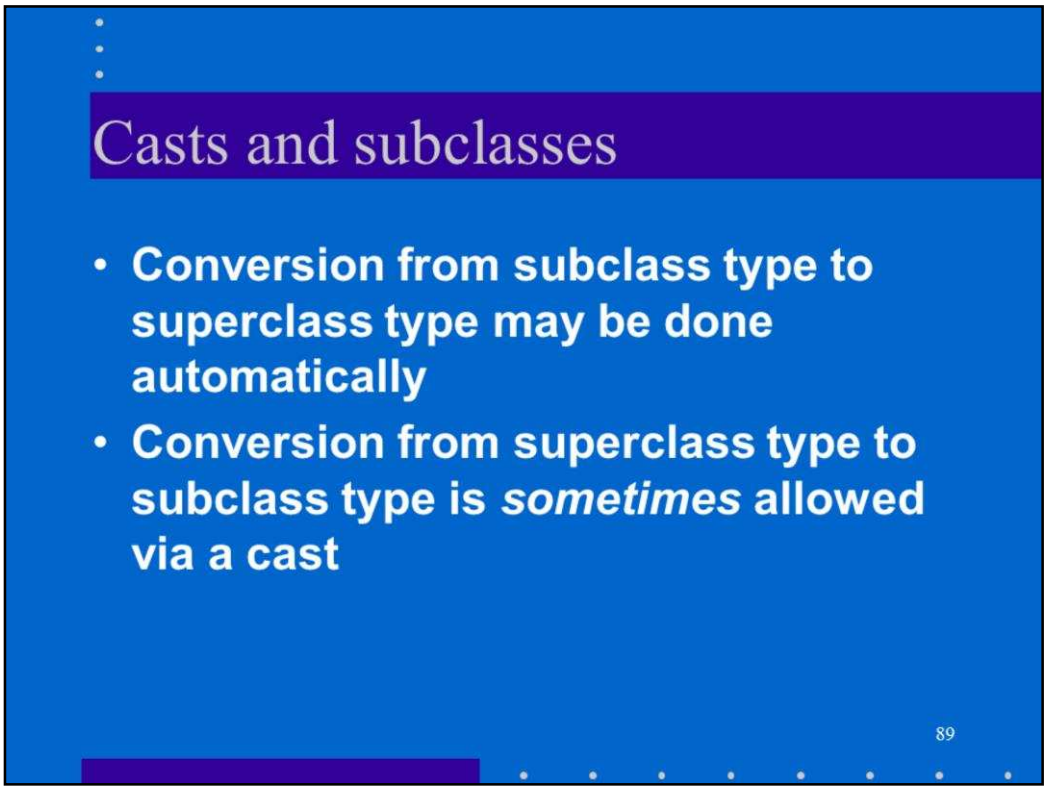
Consequently given the above object creation, the statement:

NumberedDoor n = (NumberedDoor) d;

is legal but only because d refers to an object which was created from the class NumberedDoor.

Such a cast is dynamic and its validity can only be determined at runtime and

not by the compiler.


Casts and subclasses
• Conversion from subclass type to superclass type may be done automatically
• Conversion from superclass type to subclass type is *sometimes* allowed via a cast
89

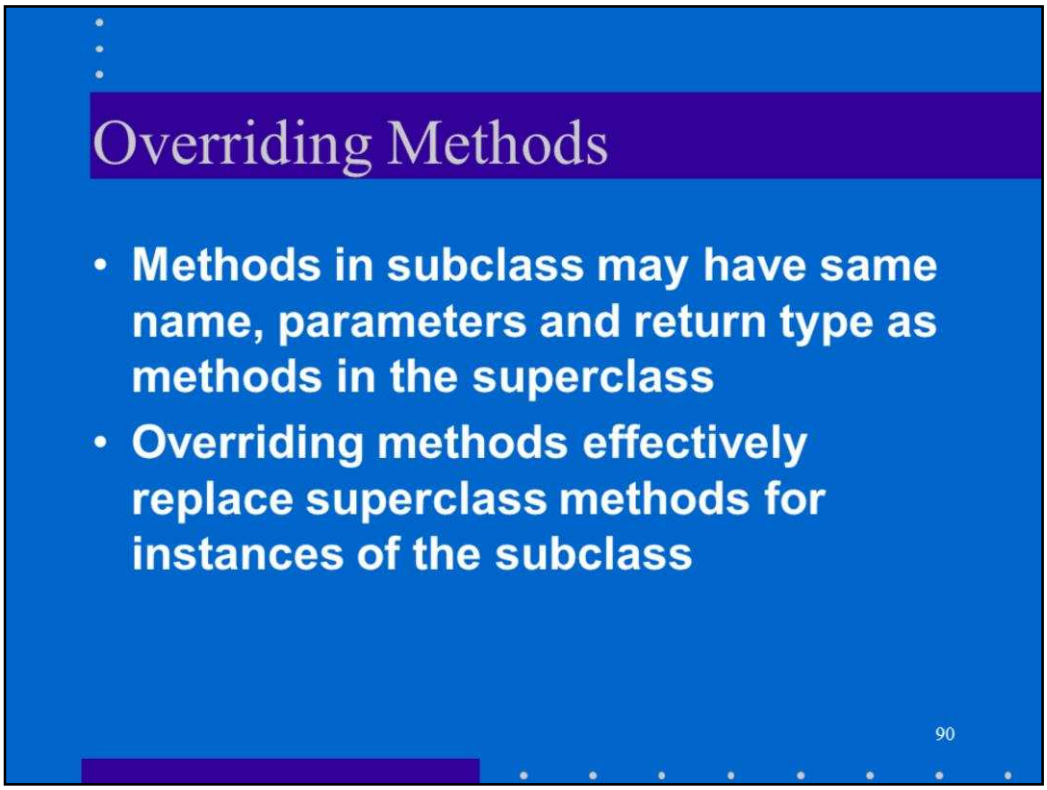
In the previous lecture a distinction was drawn between automatic type conversion by the Java compiler and explicit type conversion via a cast operation.

A subclass is a specialised version of a superclass and so type conversion from a subclass to its superclass is always safe. The Java compiler, therefore provides automatic type conversion from a subclass instance to a suitable superclass instance.

Conversion from a superclass to a subclass may not always work. Suppose we have class Vehicle with subclasses Van and Car. A Car is a Vehicle. A Van is a Vehicle. But a Car is not a Van and a Van is not a Car.

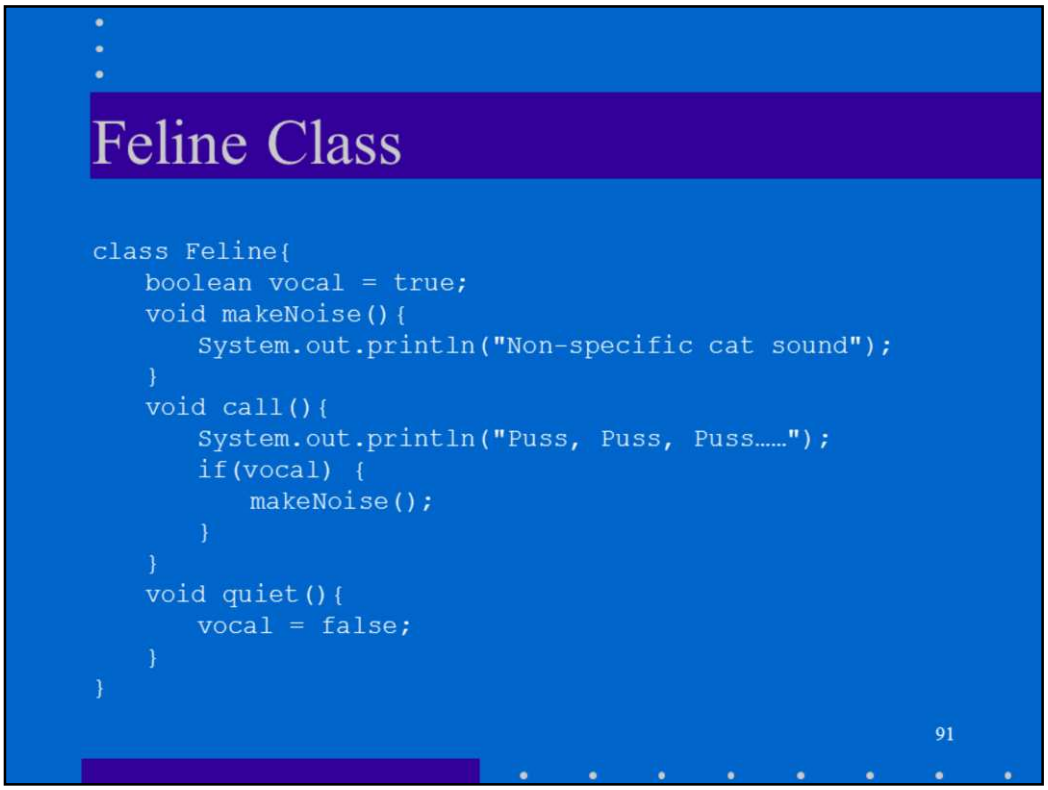
Suppose we also have an object reference variable `x` which has type `Vehicle`, this means that it can reference a `Vehicle` object. Consequently, it can reference both `Car` and `Van` objects. If we attempt to cast the type of variable `x` to a `Van` reference the cast will only work if, at the time the cast takes place, `x` is referencing an instance of a `Van` object.

This sort of cast can only be checked at run time. If, during the execution of a program, an attempt is made to cast `x` to a `Van` reference when it is actually referencing a `Car` object, an exception (see later) will be thrown.


Overriding Methods
• Methods in subclass may have same name, parameters and return type as methods in the superclass
• Overriding methods effectively replace superclass methods for instances of the subclass
90

It maybe that a subclass wishes to implement a method which has the same name as a method in its superclass but which operates in a different way. For example, suppose we were able to draw an instance of a `Door`. For a simple `Door` the draw method would draw it either open or closed, for a `NumberedDoor` the picture would include the number of the door as well as whether it was open or closed. In both cases we would send the object a draw message but the outcomes would be different. `NumberedDoor` would need a different version of `draw()` to `Door`. The return value of `draw()` and its parameters would be identical for both classes.

This is called *overriding*. Each class would have its own version of `draw()`. A `Door` object when sent a draw message would use the `Door` version of `draw()`, whilst a `NumberedDoor` object would use its own version.



This example is adapted from the Java Sourcebook, E.Anuff, Wiley, 1996. ISBN:0-4671-14859-8.

Here is a class `Feline`. A `Feline` can be called. When it is called if the `Feline` is a vocal cat it makes a noise. Since a `Feline` could be a lion, tiger, leopard or any other member of the cat family we don't know exactly what that sound will be, the best we can say is that it makes a cat sound.

By default `Felines` are vocal but we can make a quiet `Feline` by invoking the method `quiet()`.

Lion and Moggy

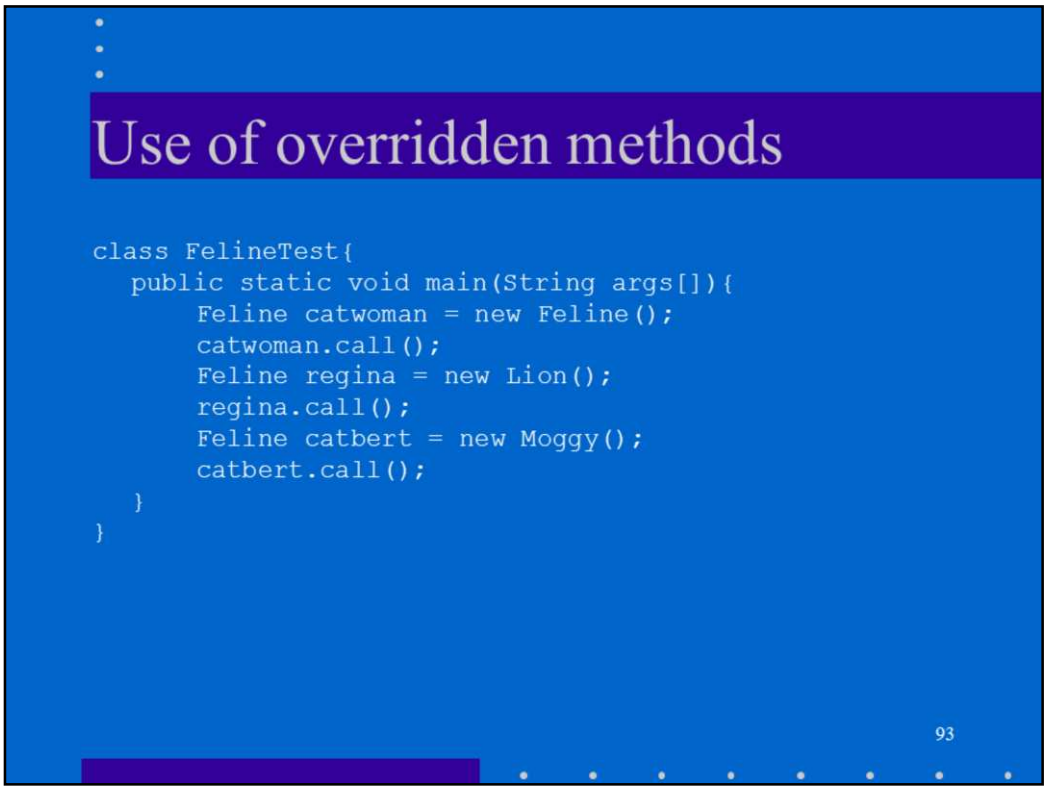
```
class Lion extends Feline{
    void makeNoise(){
        System.out.println("Roar! Roar!");
    }
}

class Moggy extends Feline{
    void makeNoise(){
        System.out.println("Meow Meow");
    }
}
```

92

Here are two subclasses of `Feline`. All these do is to override the definition of `makeNoise()`. A `Lion` is a `Feline` that roars. A `Moggy` is a `Feline` that meows.

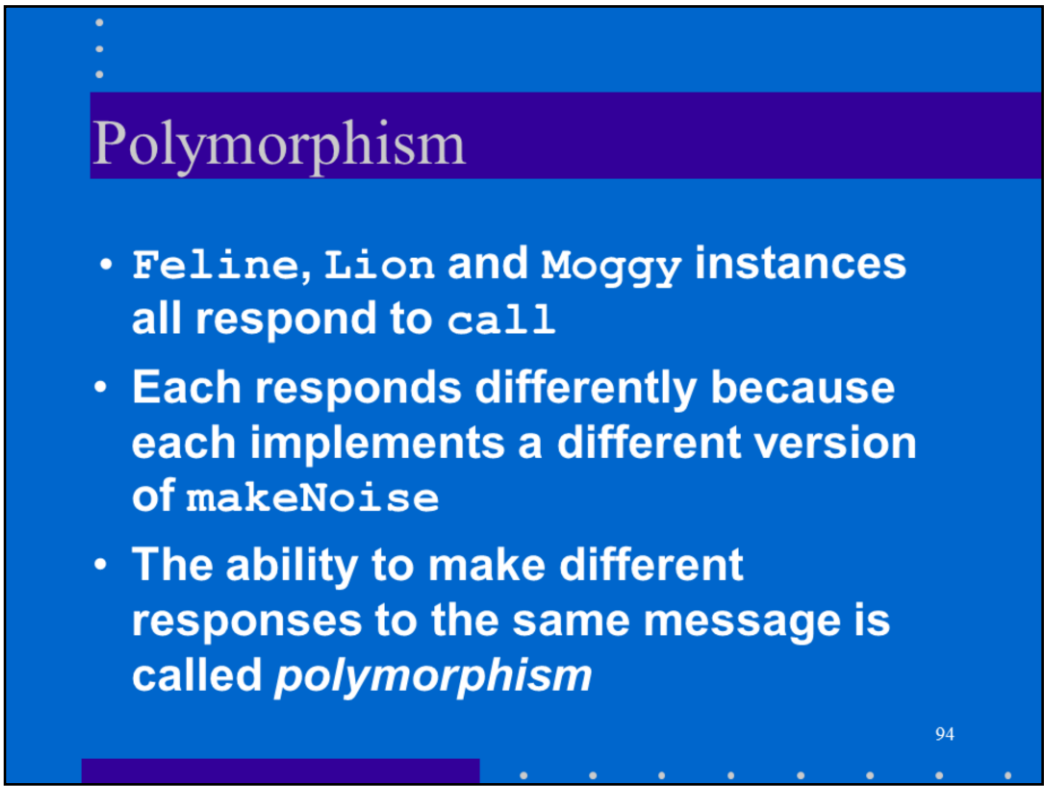
Note that the name, the parameters and the return type of the overridden function is the same in all three classes.



We can test these classes out with the code shown on the slide.

All the object reference variables, catwoman, regina and catbert hold are of type `Feline`, i.e. they hold references to `Feline` objects. But the objects created belong to different classes. The class is specified when the object is created. The effect of `call()` depends on the class the object belongs to and not the type of the variable used to reference it.

The result will be that `catwoman` will make a non-specific cat noise, `regina` will roar and `catbert` will meow.


Polymorphism
• Feline, Lion and Moggy instances all respond to call
• Each responds differently because each implements a different version of makeNoise
• The ability to make different responses to the same message is called polymorphism
94

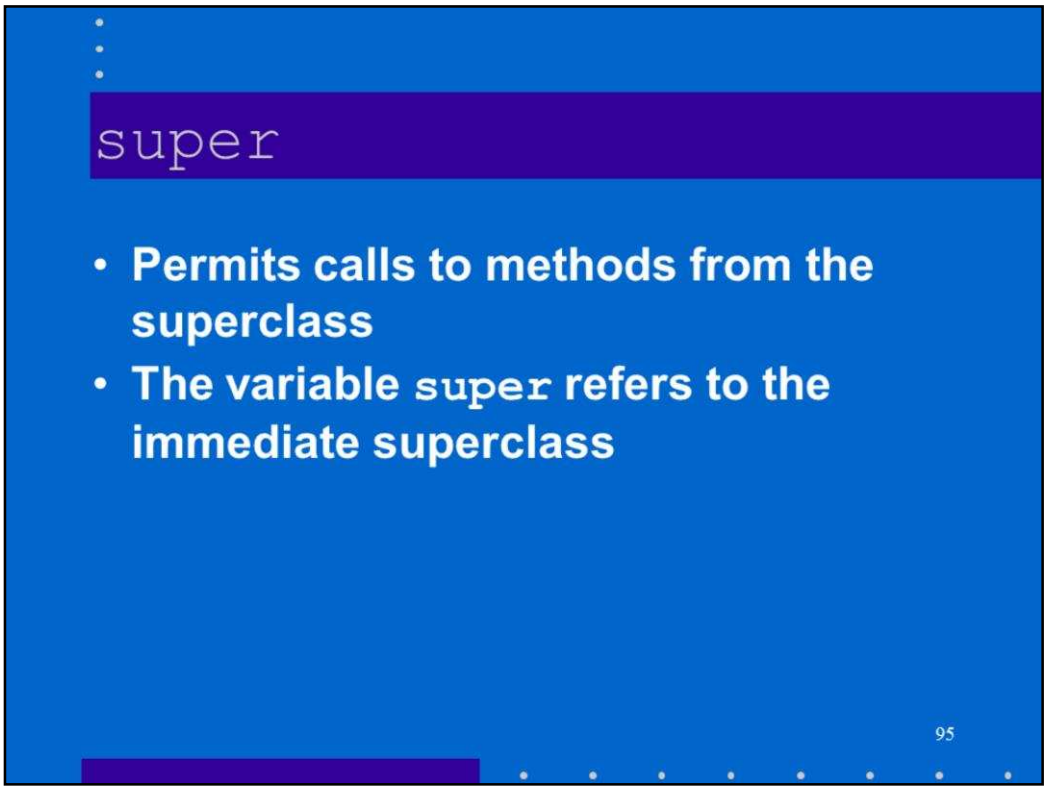
`call()` is inherited so `Feline`, `Moggy` and `Lion` will all respond to it.

Each of these classes has their own version of `makeNoise()` so in each case the response is different.

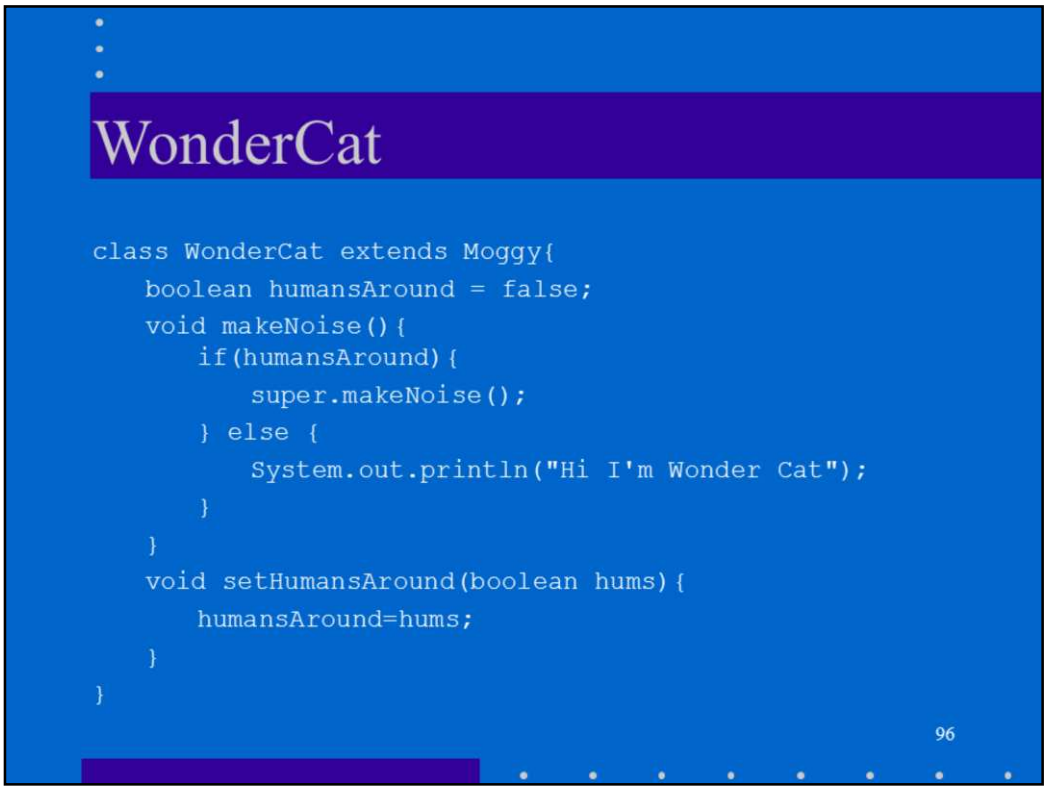
In the example code though for all three invocations of call the type of the object reference is `Feline`. So applying the same message to three different objects of the same type has produced three different outcomes.

The ability of objects of the same type to perform different tasks in response to the same message is known as *polymorphism*. This is an extremely useful feature of Java.

C++ exhibits polymorphism but in a much less natural fashion.



When writing methods for a subclass it is often useful to refer to methods in the superclass. We can't do this by name alone as overriding means that methods in the subclass and superclass share a name. The special variable super allows references to methods in the immediate superclass of a class.

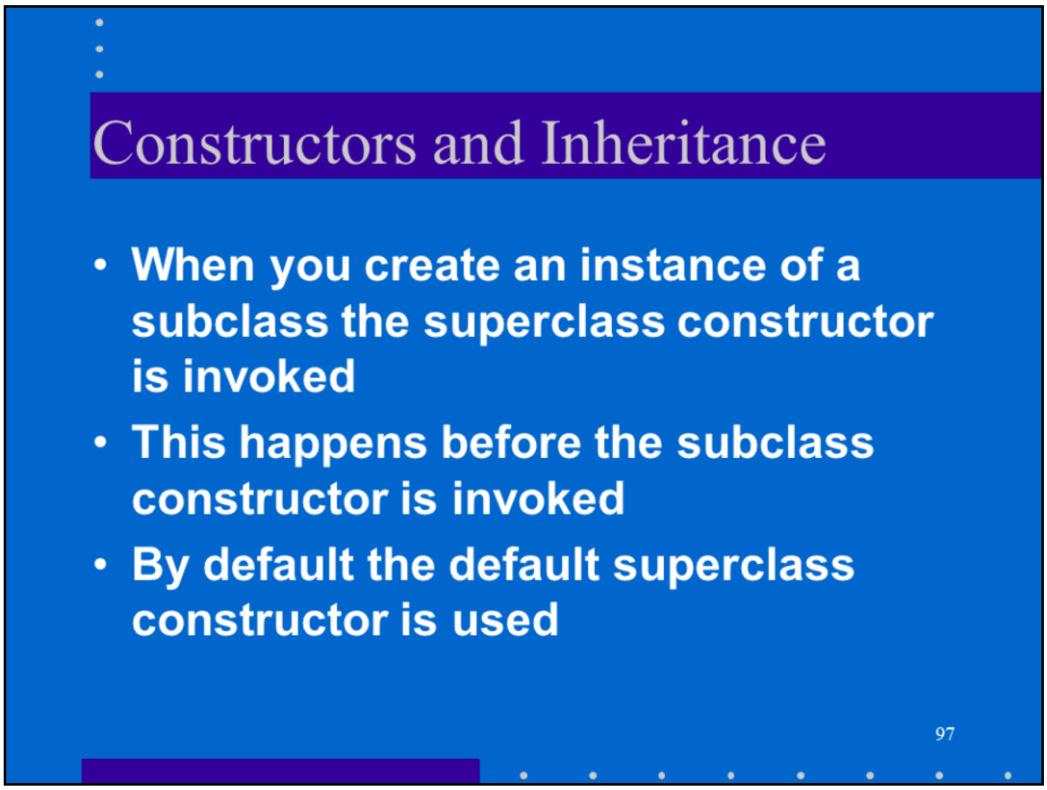


`WonderCat` is a special type of `Moggy` which can talk in English. But it only does this if there are no humans present. If humans are present it disguises itself as an ordinary `Moggy`.

As it's a special type of `Moggy` then its superclass is `Moggy`. When humans are around (`humansAround` is equal to true) then it needs to use the `Moggy` version of `makeNoise()`. To do this the program calls `super.makeNoise()` which invokes the superclasses' version of `makeNoise()`.

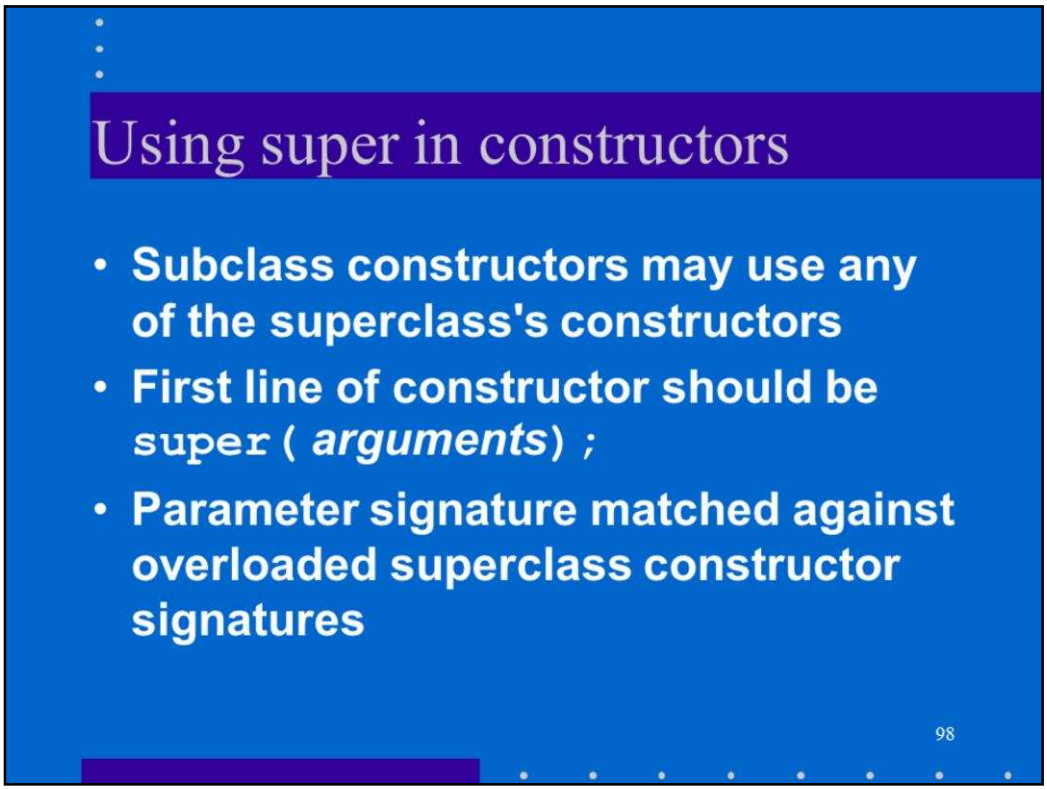
This feature is most often used when a subclass adds functionality to a superclass. Often in an overridden method you want to do everything the superclass method did and then some more. So to code this, simply use `super` to call the superclass method and then add code to do the additional tasks.

```
void overriddenMethod(){
          super.overriddenMethod()
          :              //additional code
          :
}
```


Constructors and Inheritance
When you create an instance of a subclass the superclass constructor is invoked
This happens before the subclass constructor is invoked
By default the default superclass constructor is used
97

When a class is instantiated a constructor is called. If a subclass is instantiated its superclass's constructor is called followed by its own constructor. If the superclass is the subclass of a third class, then the third class's constructor is called first, followed by the immediate superclass's constructor and finally the subclass's constructor. And so on.

Suppose any of these superclasses has more than one constructor? Which constructor will be called? If no special steps are taken then the answer is the default constructor.

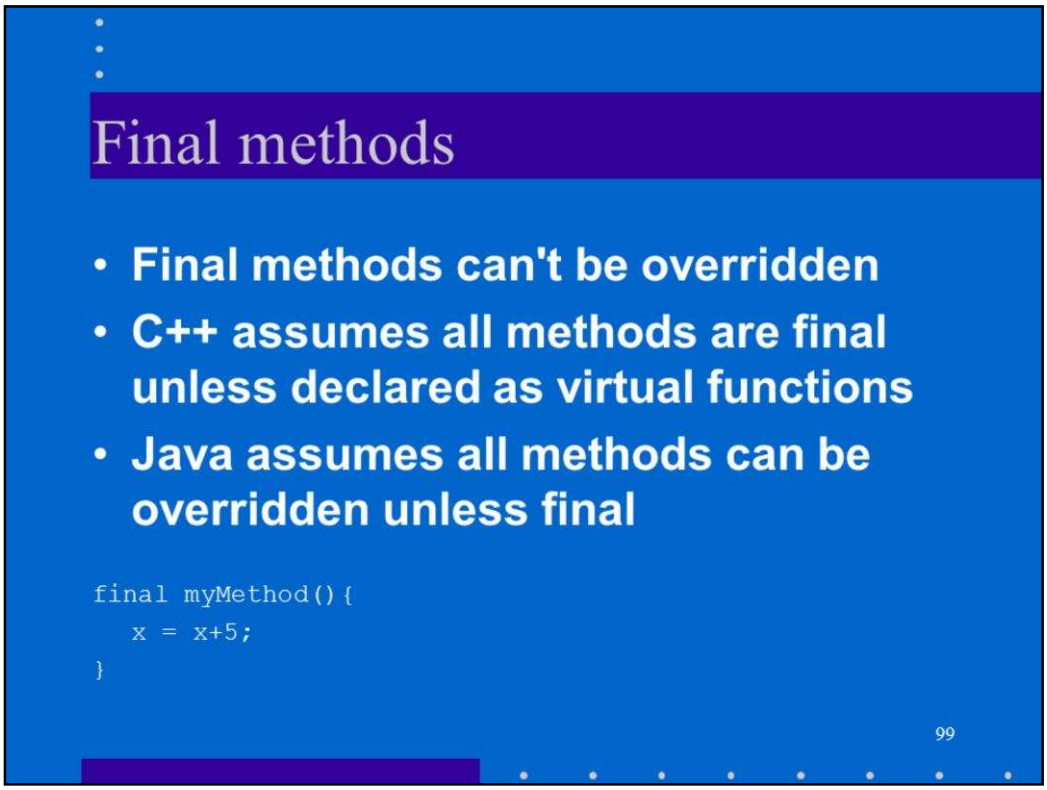

Using super in constructors
• Subclass constructors may use any of the superclass's constructors
• First line of constructor should be super (arguments) ;
• Parameter signature matched against overloaded superclass constructor signatures
98

A subclass constructor can specify which constructor should be invoked prior to its invocation. It does this by referencing super() in its first line. It has to be the very first line of the constructor. The line should read:

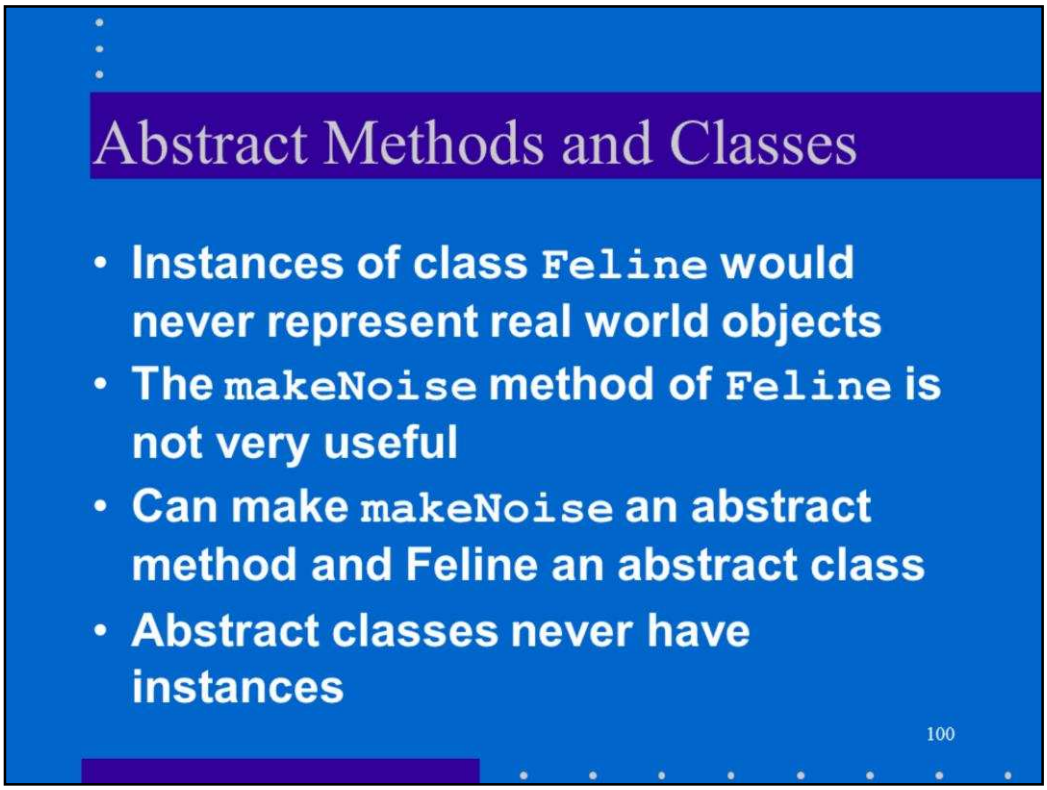
`super(`*argument list*`);`

Java will use the argument list to find the appropriate constructor, as it would do for any other overloaded method reference.

This again is useful where a subclass adds functionality to a superclass. Typically the subclass will want to implement a constructor which initialises all the variables of the superclass, and then any new variables introduced by that subclass. This can be achieved by calling an appropriate superclass constructor and then adding code to set the subclass variables.


Final methods
• Final methods can't be overridden
• C++ assumes all methods are final unless declared as virtual functions
• Java assumes all methods can be overridden unless final
final myMethod(){
x = x+5;
}
99

It is possible to prevent methods being overridden in a subclass by prefixing their definition in a class with the keyword `final`.


Abstract Methods and Classes
• Instances of class Feline would never represent real world objects
• The makeNoise method of Feline is not very useful
• Can make makeNoise an abstract method and Feline an abstract class
• Abstract classes never have instances
100

The class `Feline` was created to act as a placeholder for data and methods shared by all cats. You can never meet a `Feline` that doesn't belong to one of the other more specific classes.

Similarly the method `makeNoise()` in the `Feline` class is not very meaningful. A non-specific cat makes a non-specific sound. It has to be in the class because it has to be overridden. If it weren't present in `Feline` then the `call()` method couldn't compile. But it doesn't do anything useful.

Rather than have something in a class which does something misleading the `makeNoise()` method in `Feline` can be declared as `abstract`. This means that it doesn't exist in `Feline` but it will be defined in a subclass. This is the equivalent of a pure virtual function in C++.

A class which has an abstract method can't be instantiated and is an abstract class. No objects can be created from the class. Classes from which objects may be created are called concrete classes.

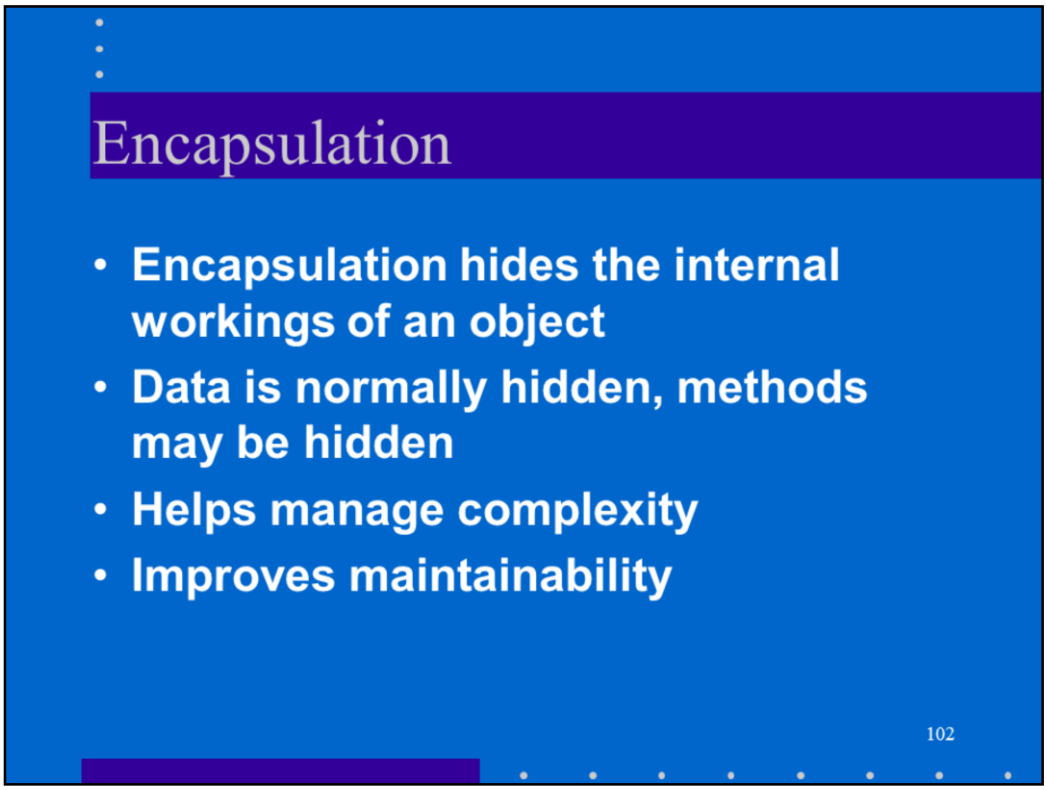
Abstract Feline

```
abstract class Feline{
   boolean vocal = true;
   abstract void makeNoise();
   void call(){
        System.out.println("Puss, Puss, Puss……");
        if(vocal)makeNoise();
   }
   void quiet(){
        vocal = false;
   }
}
```

101

Here the class definition of `Feline` and the definition of the `makeNoise()` method have been prefixed with the keyword `abstract`. `makeNoise()` no longer has any statements associated with it.

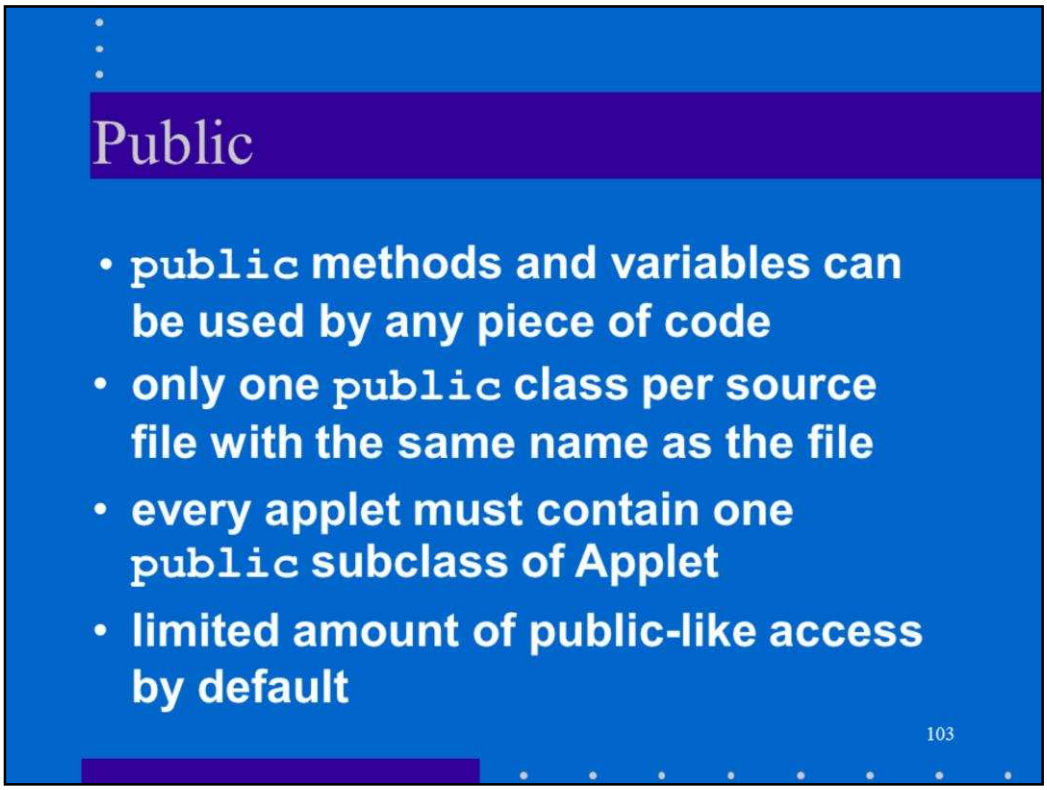
Using the `new` operator on `Feline` will result in an error as `Feline` is now an abstract class.


Encapsulation
• Encapsulation hides the internal workings of an object
• Data is normally hidden, methods may be hidden
• Helps manage complexity
• Improves maintainability
102

One aspect of programming that has become apparent over the last twenty years is that the quality of software is higher when the interface between different parts of a program is well defined. Encapsulation, which is the hiding of the internals of one part of a program from another part of the program, therefore improves software quality.

In object oriented programming objects normally make a subset of their methods available to other objects and hide away their data from each other.

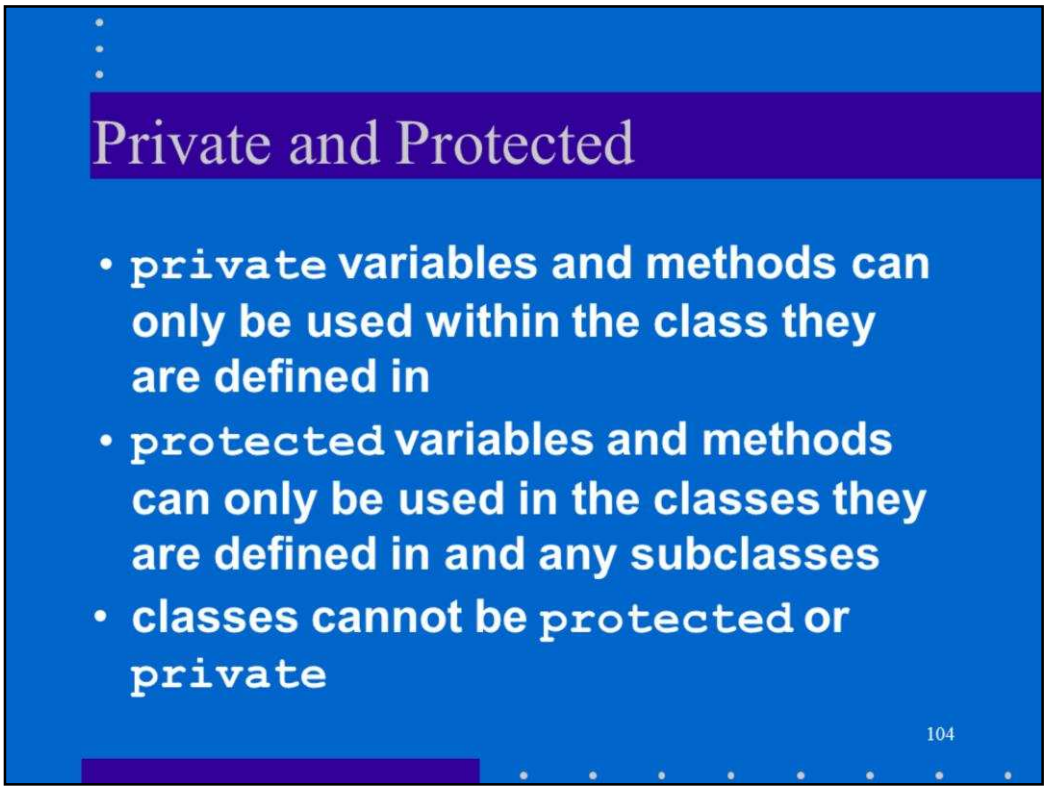
This leads to programs which are less complex and therefore easier to understand and also to software which is more maintainable because it is easier to locate code responsible for a given task.


Public
• public methods and variables can be used by any piece of code
• only one public class per source file with the same name as the file
• every applet must contain one public subclass of Applet
• limited amount of public-like access by default
103

Java aids encapsulation by introducing keywords (`public`, `private` and `protected`) to limit the visibility of data and methods.

`public` data and methods are visible to every other object, that is, any piece of code may call the methods or modify the data.

Data and methods which are neither `public`, `private` or `protected` are visible to objects created from classes in the same package (see later) that they belong to.


Private and Protected
• private variables and methods can only be used within the class they are defined in
• protected variables and methods can only be used in the classes they are defined in and any subclasses
• classes cannot be protected or private
104

Anything declared as `private` can only be used within the class it is defined. Even subclasses cannot use it.

Methods and variables which are `protected` can be referenced by code in their own class or any of their subclasses.

In general data items will be `protected` or `private` whilst methods will be `protected` or `public`.

It is good practice to restrict the visibility of a variable or method as much as possible.

Classes may be `public` but not `protected` or `private`.

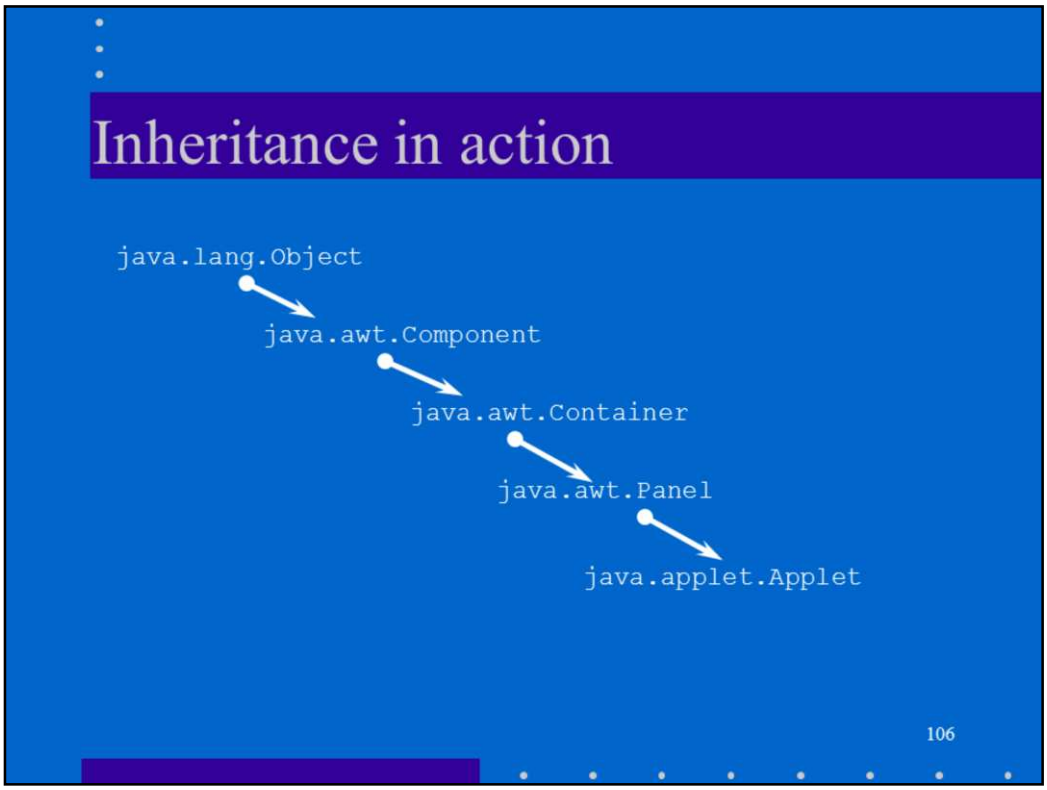
Encapsulation Example

```
abstract class Feline3{
   private boolean vocal = true;
   protected abstract void makeNoise();
   public void call(){
        System.out.println("Puss, Puss, Puss.....");
        if(vocal) {
              makeNoise();
        }
   }
   public void quiet(){
        vocal = false;
   }
}
```

105

In class `Feline` we only want to alter `vocal` within the class so it can be `private`. Even `Moggy` or `Lion` can't alter it directly, however they can call the public method `quiet()` to change it. In this way if we decide to implement recording whether a cat is vocal or not in a different way, we can hide the changes from programs that use the class. We can simply change the internals of `quiet()`.

`call()` is another public method but `makeNoise()` is not to be invoked outside of `Feline` and `Lion` and `Moggy`. We make it `protected` so that all three classes can call it.



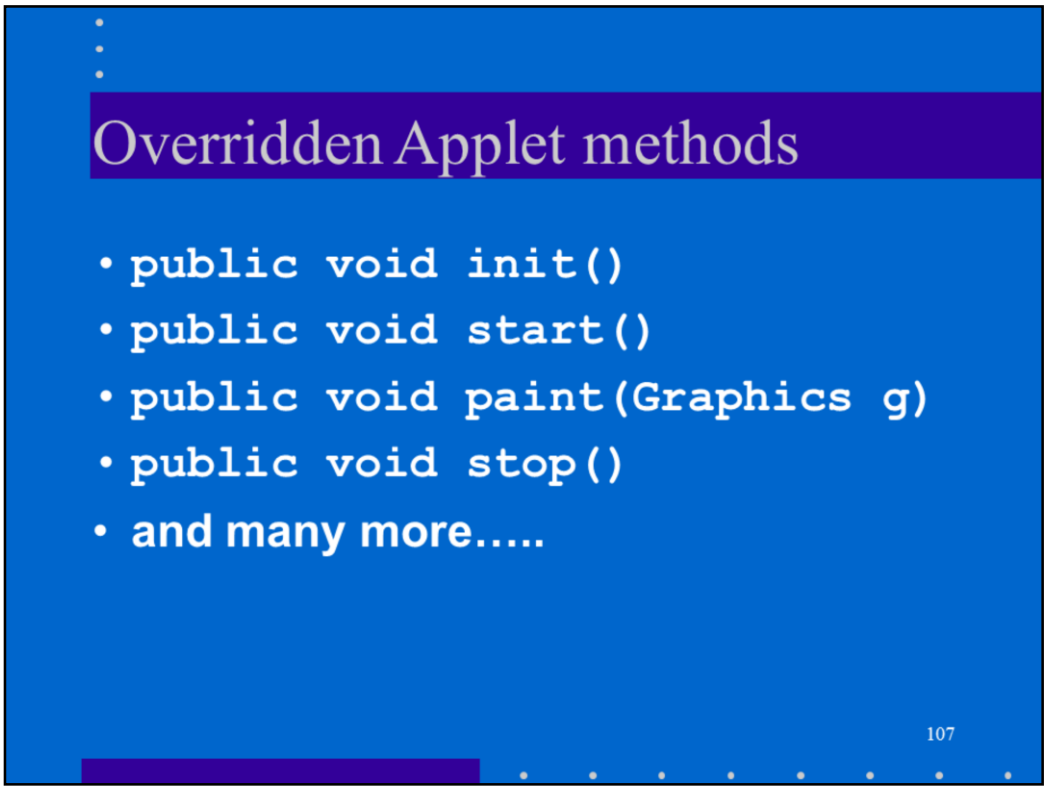
Applets (Java programs which run in browsers) all extend class `Applet`. Class `Applet` defines all the methods which are necessary for an applet.

An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application. The `Applet` class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The `Applet` class provides a standard interface between applets and their environment.

`Panel` is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.

A generic Abstract Window Toolkit (AWT) container object is a component that can contain other AWT components.

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface. The `Component` class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components.



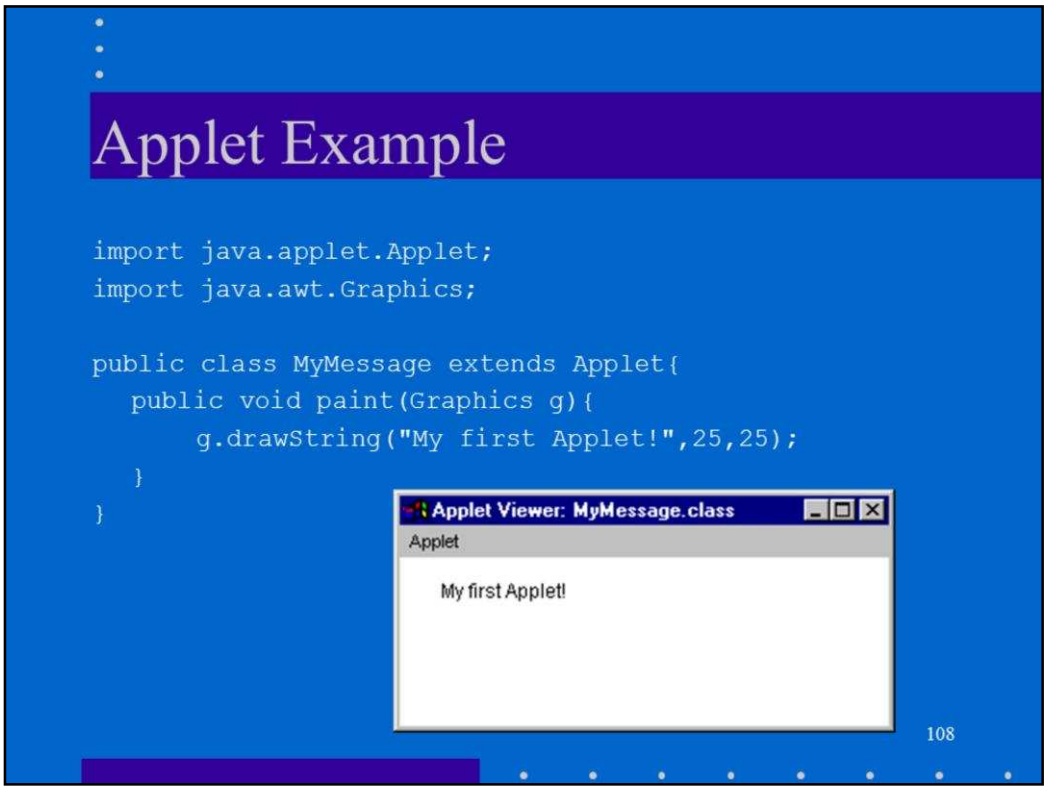
Applets are constructed by overriding methods in `Applet`. `Applet` is a concrete class so if the methods are not overridden it provides default behaviour. The methods of an applet are called by the browser.

`init()` is called by the Web browser before the applet is run. This is overridden in order to initialise instance variables in an applet. It is typically used to obtain parameters from the HTML file.

`start()` is called every time the web page containing the applet is loaded into the browser. Overloading this method controls what happens each time the applet is on the screen.

`stop()` is called every time the web page is unloaded. If an applet was making a continuous noise then `stop()` might be overloaded so as to turn the sound off when the page containing the applet was replaced in the browser.

The browser will request the applet to redraw itself periodically. This will happen following events that occur in the windowing environment in which the browser is running. The browser causes the applet to redraw itself by sending the `paint()` message. Overloading the `paint()` message will therefore alter what appears on the screen. The browser passes a `Graphics` object to `paint()`. This `Graphics` object is an object which represents the part of the screen on which the applets displays. Messages sent to the object alter what is displayed.



This applet makes use of the `Applet` class and the `Graphics` class. As these classes are in packages provided with the JDK they must be imported. The `import` statement does this.

Class `MyMessage` inherits from (extends) `Applet`.

Most of the things required in this applet are already provided by class `Applet`. It is only necessary to override one method: `paint()`.

The code which overrides a method must have the same signature as the method it overrides. The *signature* of a method consists of the name of the method and the number and types of formal parameters to the method. The access modifier of an overriding or hiding method must provide at least as much access as the overridden or else a compile-time error occurs. If a method declaration overrides the declaration of another method, then a compile-time error occurs if they have different return types or if one has a return type and the other is void. So the version of `paint()` in `MyApplet` must be `public`, must return void and must have a single parameter of type `Graphics`.

The overridden version of `paint()` will be called by the browser whenever necessary. All it does is ask the `Graphics` object passed to it to display a string at a certain position within the applet's allotted space in the browser.

The message `drawString` when sent to a `Graphics` object takes a `String` parameter and two `int`s. The method it invokes displays the `String` at the position specified by the integer parameters.

HTML file

```
<html>
<head>
</head>
<body>
<applet code="MyMessage.class" width=300 height=100>
</applet>
</body>
</html>
```

109

`MyApplet.java` is compiled to generate `MyApplet.class`. This is then referenced in the `<Applet>` tag of the HTML file.

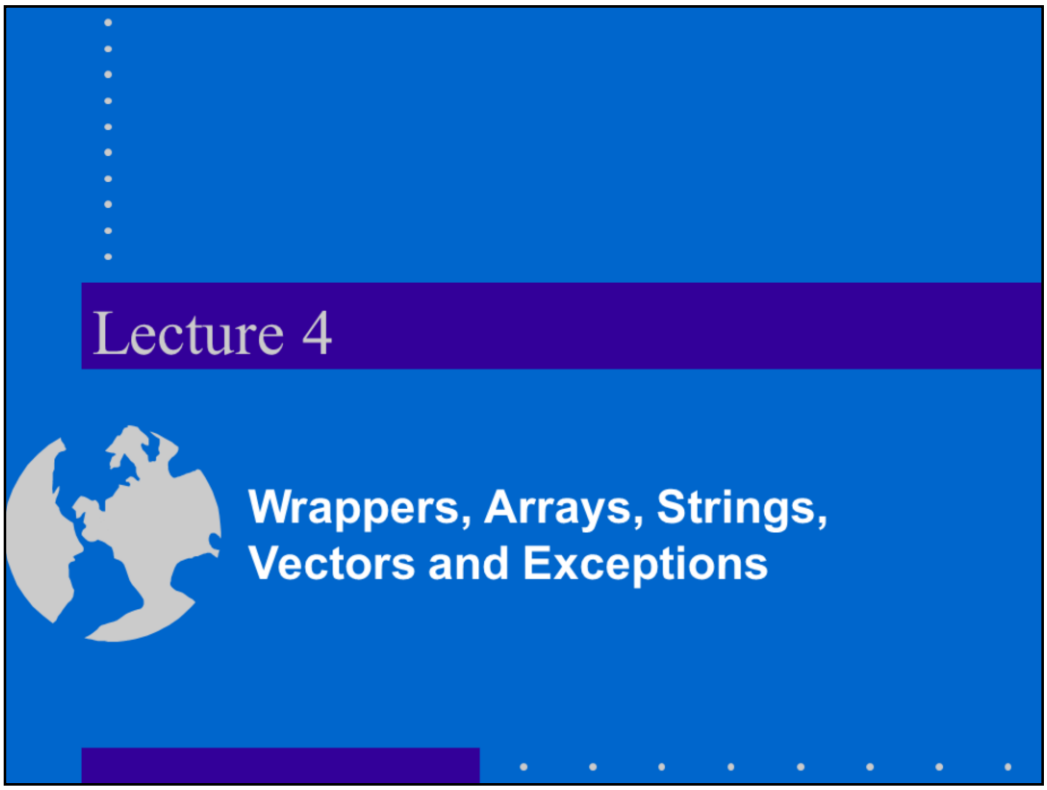
When the applet is loaded the browser will call `Applet`'s version of `start()`. This will call a method called `repaint()` which will result in a call to `paint()`. Since MyApplet overloads `paint()` its version of `paint()` will run rather than `Applet`'s default version.

This will write out the string "My first Applet".

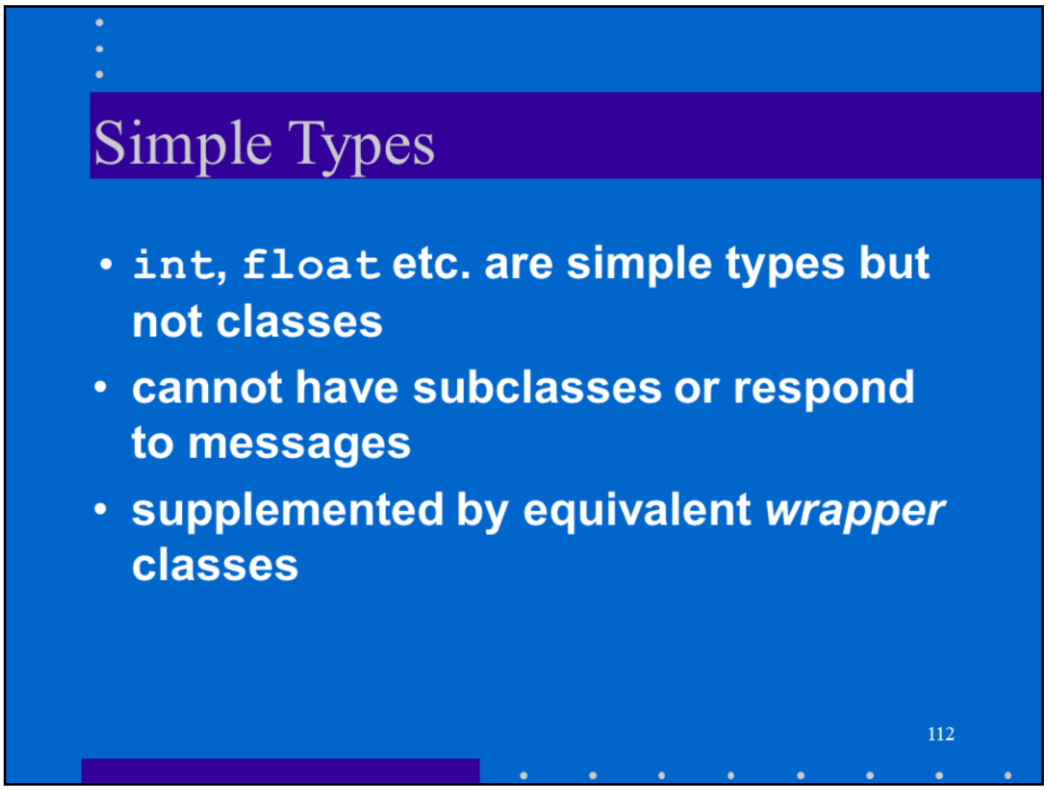
The browser will call `paint()` each time it is resized or moved and so the string will be rewritten whenever necessary.

Summary

- Inheritance
- Subclasses
- Final classes
- Overriding Methods
- Final Methods
- `super`
- Abstract methods and classes
- Encapsulation



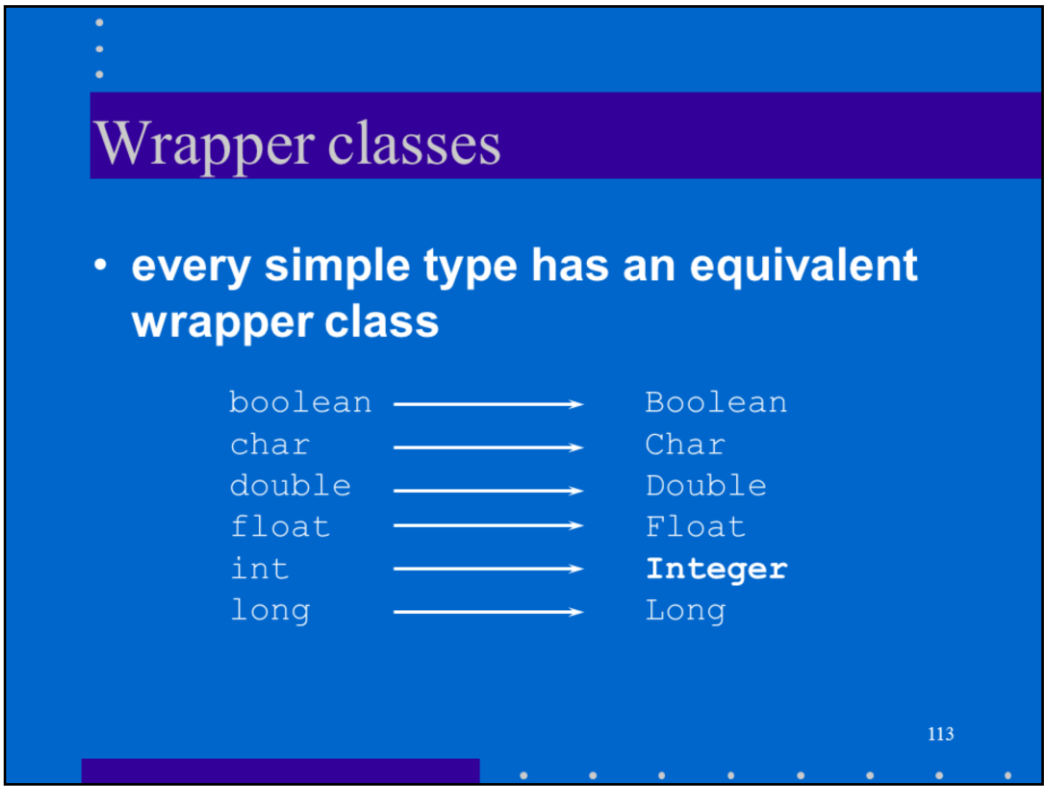
This lecture deals with a number of topics which are largely independent of each other but are all necessary knowledge for Java developers.


Simple Types
• int, float etc. are simple types but not classes
• cannot have subclasses or respond to messages
• supplemented by equivalent wrapper classes
112

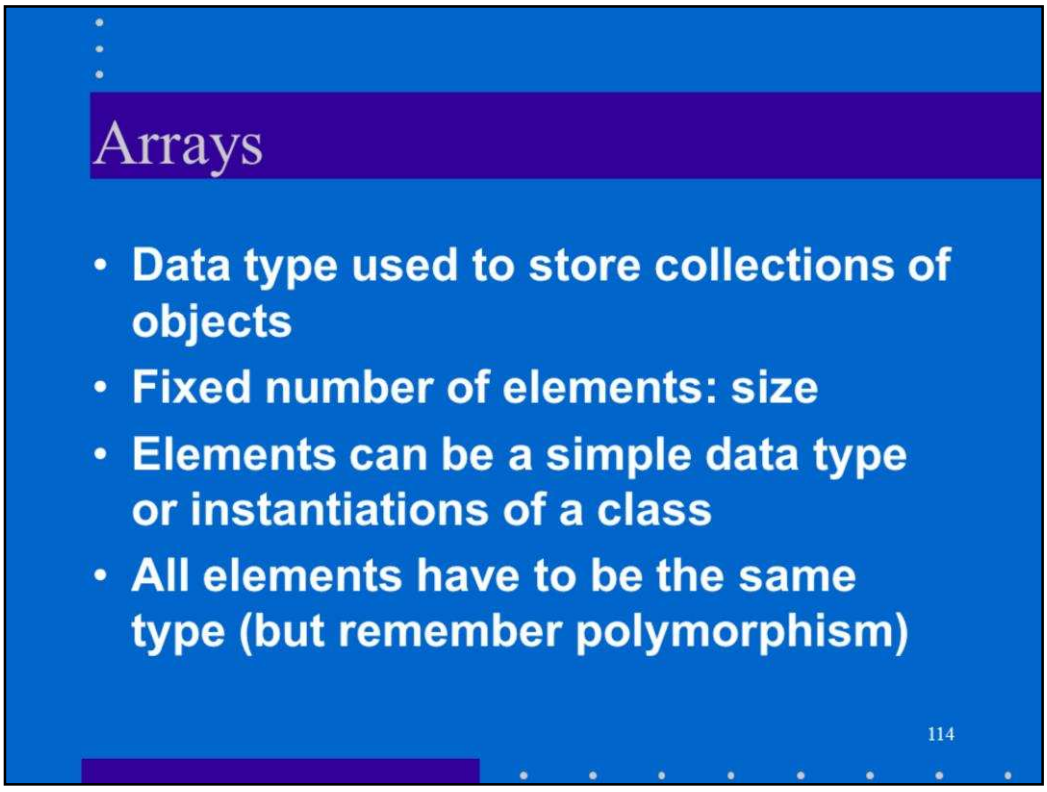
In pure object-oriented languages such as Smalltalk everything is an object. Java takes a pragmatic approach and uses built-in types in the same way as C and C++ does. This means that variables of type `int`, `float` etc. are not objects. This means that it isn't possible to have a class which is a subtype of an `int` or a `float`. It also means that variables holding `int`s or `float`s do not respond to messages and can only be manipulated by operators (e.g. `+`, `*`).

It is conceivable that we might want a special sort of `int`, for example, `PositiveInt` (which cannot have negative values). Also some classes will provide methods which have parameters that must be object references (i.e they expect to be able to apply some of the methods contained in `Object` to an argument). The methods will therefore not accept variables which have been declared as a basic type.

To deal with these problems Java provides a set of classes which are equivalent to the basic types. They are called wrapper classes because they take a basic type and embed it into an equivalent class. These classes can yield objects that to all intents and purposes behave like variables of the basic types. Unlike variables of the basic types, however, you can develop specialised versions of them through inheritance and they understand messages.



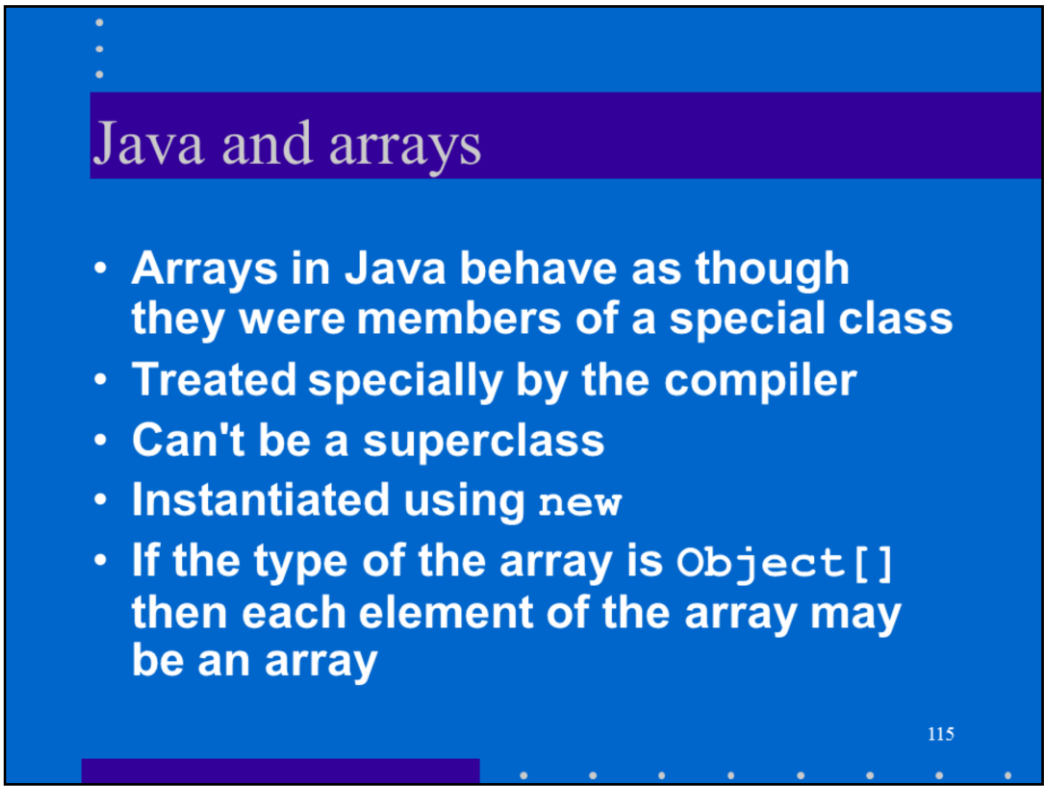
The mapping from basic types to their equivalent class representations is fairly straightforward. Each basic type has an equivalent wrapper class with the same name as the type but with the first letter in its name capitalised. All types, that is except `int` whose wrapper class is `Integer` not `Int`.


Arrays
• Data type used to store collections of objects
• Fixed number of elements: size
• Elements can be a simple data type or instantiations of a class
• All elements have to be the same type (but remember polymorphism)
114

As in C++ an array is a data structure which will hold a number of elements of the same type. As with arrays in C++, arrays in Java have a declared size which cannot be altered.

Arrays in C++ and Java may contain either object references or variables of the basic types.

Whilst an array must contain elements of a declared type, remember that via polymorphism these elements may be from a number of different subclasses of the class which defines the type of the array. So for example if the class `Feline` from Lecture 3 is used in the declaration of an array, objects in that array could be `Lion`s, `Moggy`s or `WonderCat`s.


Java and arrays
• Arrays in Java behave as though they were members of a special class
• Treated specially by the compiler
• Can't be a superclass
• Instantiated using new
• If the type of the array is Object[] then each element of the array may be an array
115

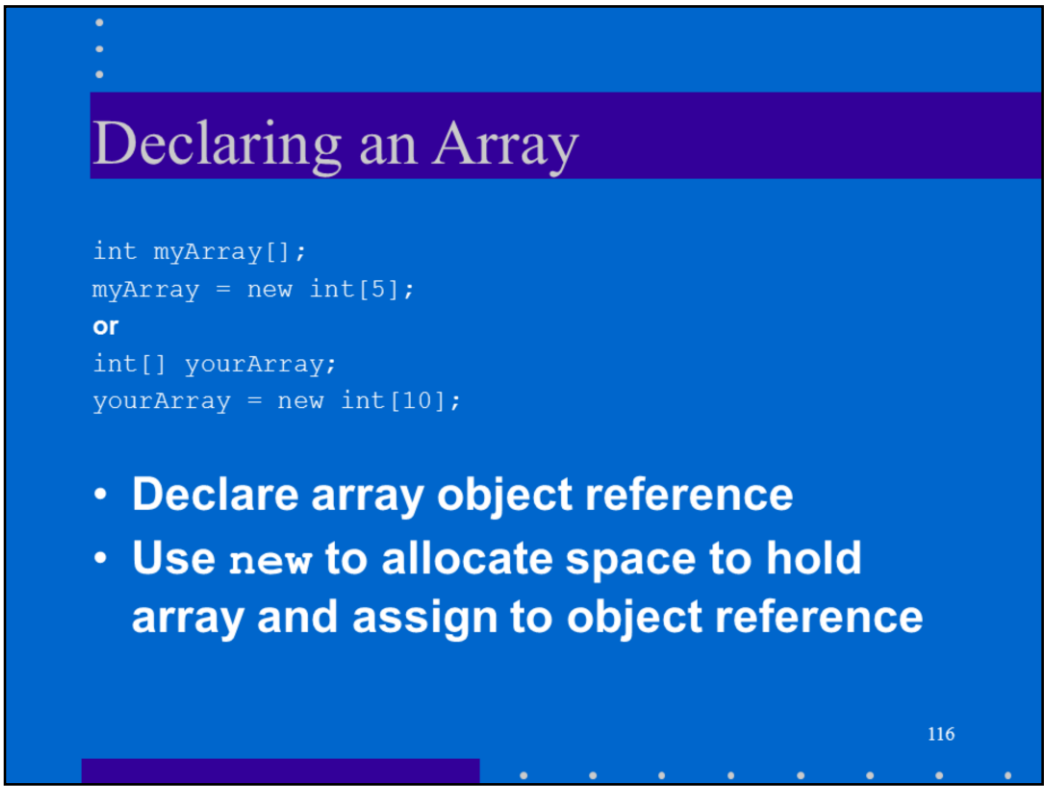
The Java specification states clearly that arrays are objects. But it doesn't say what class they belong to! So you can't have a specialised version of an array. As an array is an object it is instantiated via the `new` operator.

The Java specification says: “Java *arrays* are objects, are dynamically created, and may be assigned to variables of type Object . All methods of class Object may be invoked on an array.”.

An array object contains a number of variables. The number of variables may be zero, in which case the array is said to be *empty*. The variables contained in an array have no names; instead they are referenced by array access expressions that use nonnegative integer index values. These variables are called the *components* of the array. If an array has *n* components, we say *n* is the *length* of the array; the components of the array are referenced using integer indices from 0 to n-1, inclusive.

All the components of an array have the same type, called the *component type* of the array. If the component type of an array is *T*, then the type of the array itself is written *T*[].

There is one situation in which an element of an array can be an array: if the element type is Object, then some or all of the elements may be arrays, because any array object can be assigned to any variable of type Object.


Declaring an Array
int myArray[];
myArray = new int[5];
or
int[] yourArray;
yourArray = new int[10];
• Declare array object reference
• Use new to allocate space to hold array and assign to object reference
116

An array is an object. To refer to an array we need to create an object reference variable of the appropriate type. Normally the type would be the same name as a class. In the case of arrays, things are slightly different. The type of an array of `int`s is `int[]`. So a variable which references an array of `int`s must be declared as `int[]`. To confuse things further Java provides two types of syntax for this type of declaration.

```
int myArray[];
```

```
int[] myArray;
```

Both of these statements declare an object reference variable which will hold a reference to an arrays of `int`s. The first uses the familiar C, C++ syntax. The second is more logical because it defines an object reference of type `int[]` in the normal Java declaration style. The second format is preferable but less familiar.

Arrays are allocated storage via the `new` operator. This requires the type of the array members and the length of the array specified in square brackets (`[]`). The result of the new operator should be stored in a variable which can hold references to array objects of the appropriate type.

Array declaration (shorthand)

- **Two versions:**

array-type array-name`[]` `=` `new` *array-type*`[`array-size`]`
array-type`[]` *array-name* `=` `new` *array-type*`[`array-size`]`

```
int myArray[] = new int[5];
int[] yourArray = new int[10];
```

- **Multidimensional arrays are declared as arrays of arrays**

117

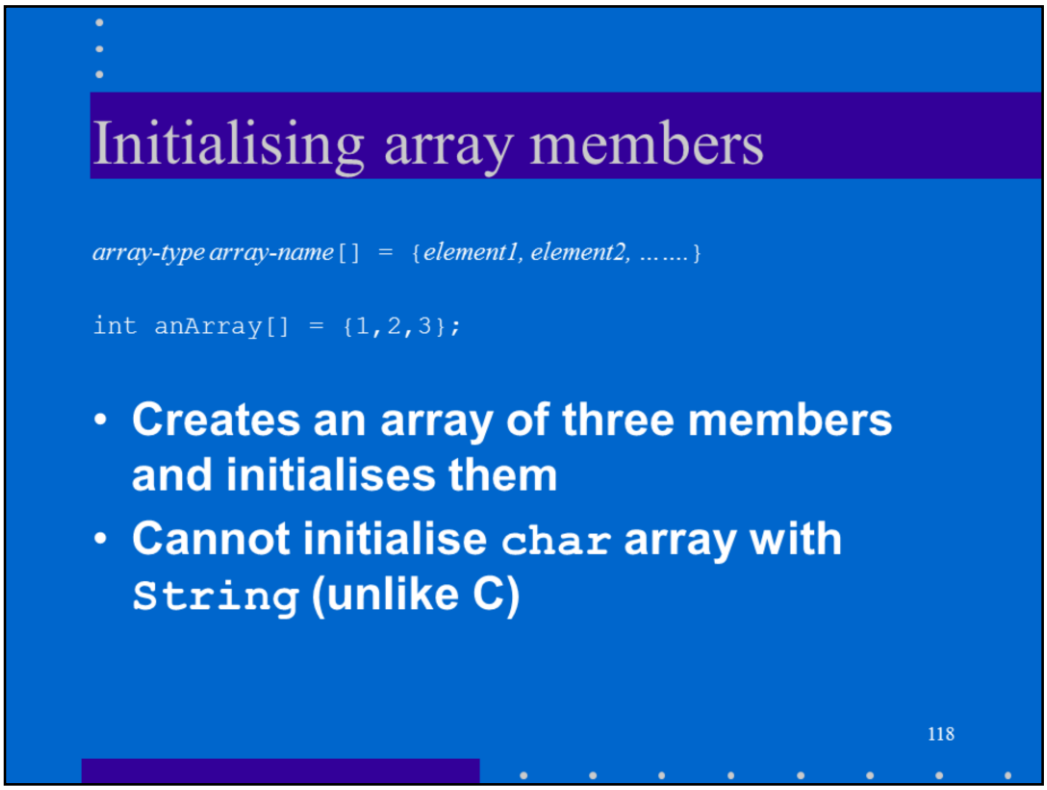
Array declaration and storage assignment can be done in a single statement. Again there are two versions.

Multidimensional arrays are created by creating an array of arrays

```
int [] [] matrix = new int [2] [2];
```

This declaration declares a 2 by 2 (4 cell) matrix of integers.

The same effect can be achieved by declaring an array where the elements are arrays.


Initialising array members
array-type array-name[] = {element1, element2, ...}
int anArray[] = {1,2,3};
• Creates an array of three members and initialises them
• Cannot initialise char array with String (unlike C)
118

Arrays can be given initial values in a manner similar to C and C++. The array will be allocated sufficient space to hold all the initialisation values. In the example on the slide a three element array will be created. The three elements will be set to 1, 2 and 3.

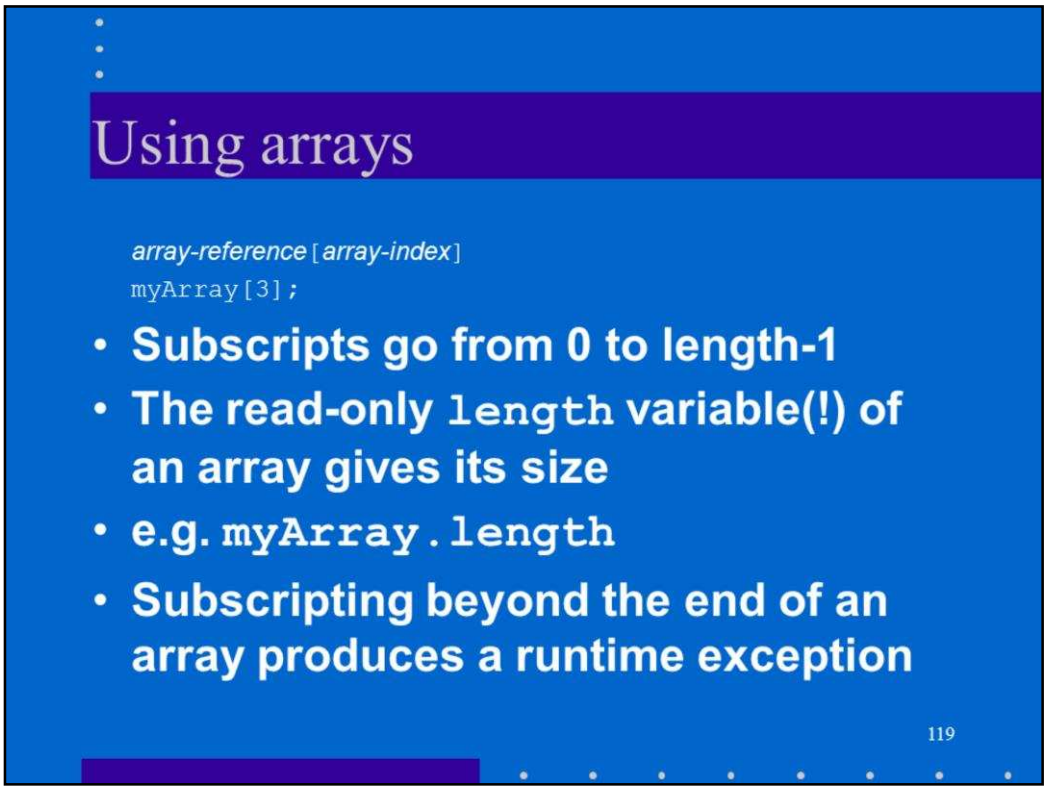
The only difference between the C, C++ mechanisms for initialising an array is that character arrays cannot be initialised using a `String` literal.

```
char[] alphabet = "abc";
```

would be flagged as an error by the Java compiler.

The correct version is:

```
char[] alphabet = {'a','b','c'};
```


Using arrays
array-reference [array-index]
myArray[3];
• Subscripts go from 0 to length-1
• The read-only length variable(!) of an array gives its size
• e.g. myArray.length
• Subscripting beyond the end of an array produces a runtime exception
119

Like C and C++ the first element in an array has a subscript of 0. If an array has 10 elements (length 10) then the subscript of the final element in the array is 10-1 which is 9.

All Java arrays have a read only instance variable called length. This can be used by a programmer to control loops which access the array.

Array elements are referenced by using the name of an object reference variable which refers to the array and a subscript (an integer in square brackets ([])).

In Java if you use a subscript which references an element beyond the end of the array then the runtime system will generate an exception (see later). C and C++, on the other hand, allow a programmer to access beyond the end of an array. This has been source of error in many programs. C programmers would argue that it is the programmer's responsibility to get things right and that if the system is testing for these errors on every array access then this is a source of inefficiency.

Array example

```
class ArrayTest{
   public static void main(String args[]){
        int[] test = {34,56,89};

        for(int i=0;i<test.length;i++)
              System.out.println(test[i]);

   }
}
```

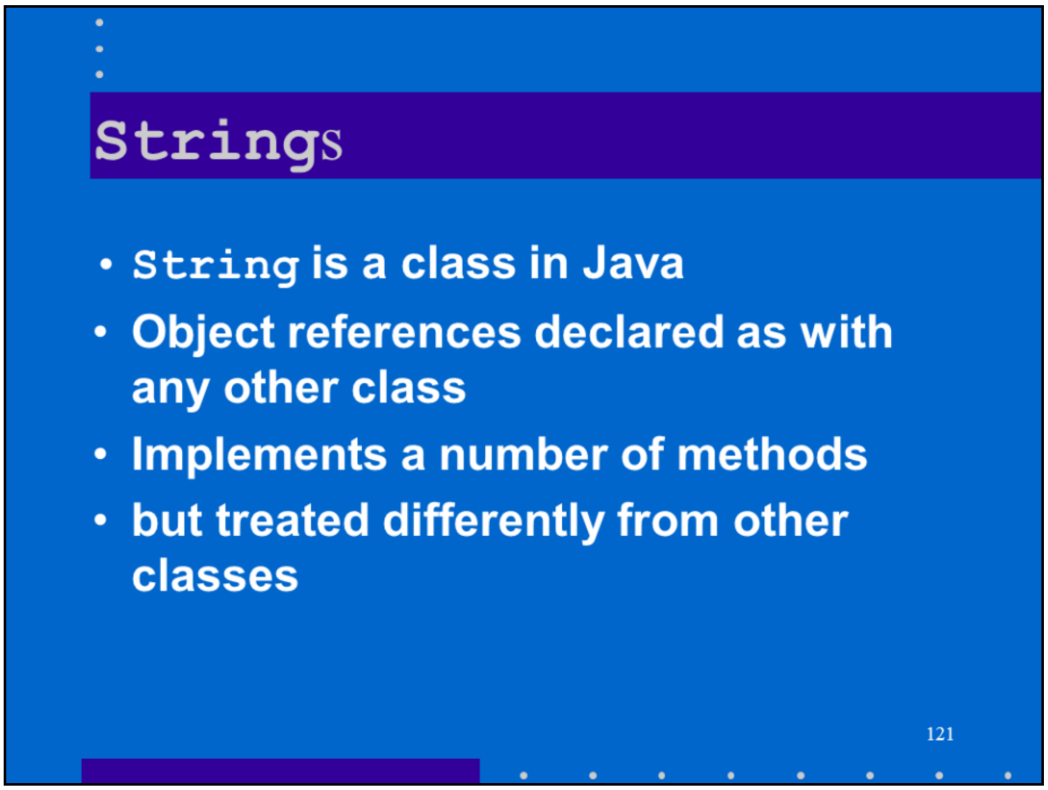
120

This program shows the use of an array.

The variable `test` is declared to be an object reference of type `int[]`.

It is initialised as an array of three integer numbers: 34,56 and 89.

The `for` loop prints out the numbers using the read-only instance variable `test.length` to control the number of iterations. `test.length` contains the number of elements in the array `test`.

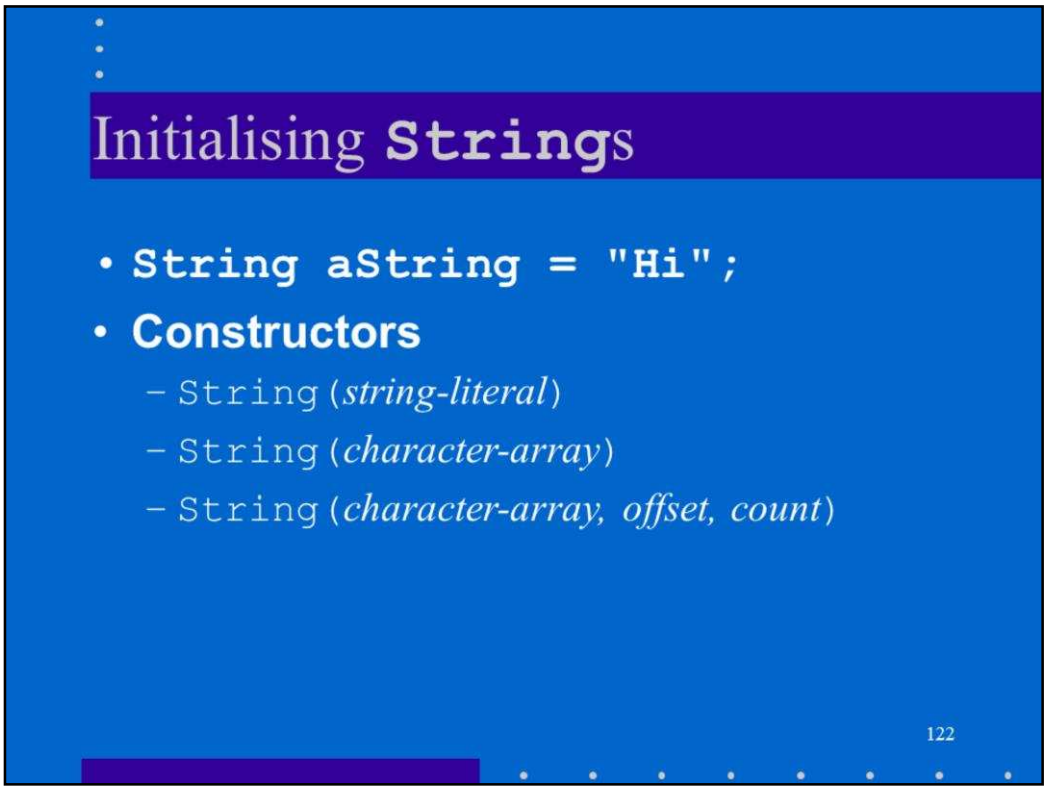

Strings
• String is a class in Java
• Object references declared as with any other class
• Implements a number of methods
• but treated differently from other classes
121

To all intents and purposes `String` appears to be a class identical to any programmer defined classes in Java.

A `String` object is created in the same way as one would create any other type of object.

Like any other class it defines a number of methods which can be applied to the class (if they are static) or to instances of class `String`.

It does, however, differ in two ways from standard classes. It can be initialised without specifying a constructor and it has its own special concatenation operator.



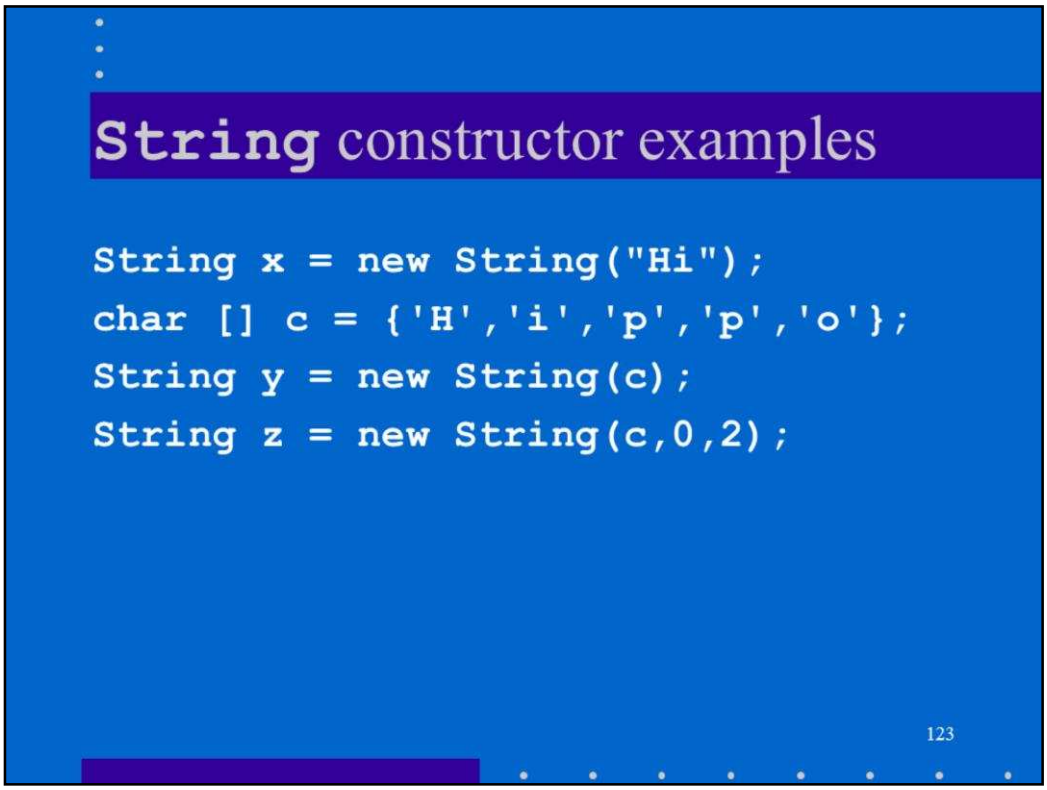
The first point on this slide shows the creation of an object reference to a `String` which is initialised to refer to a `String` containing `"Hi"`. Peculiarly, this is achieved without the use of the `new` operator.

There are also several constructors for initialising `String`s. These are used in conjunction with the `new` operator.

The first shown on the slide initialises the `String` to the value of the string literal passed and an argument.

The second effectively converts a character array into an equivalent `String`.

The third takes a number of contiguous characters from a character array and creates an equivalent `String`. (In this context, contiguous means that the elements are next to each other in the array).




```
String x = new String("Hi");
```

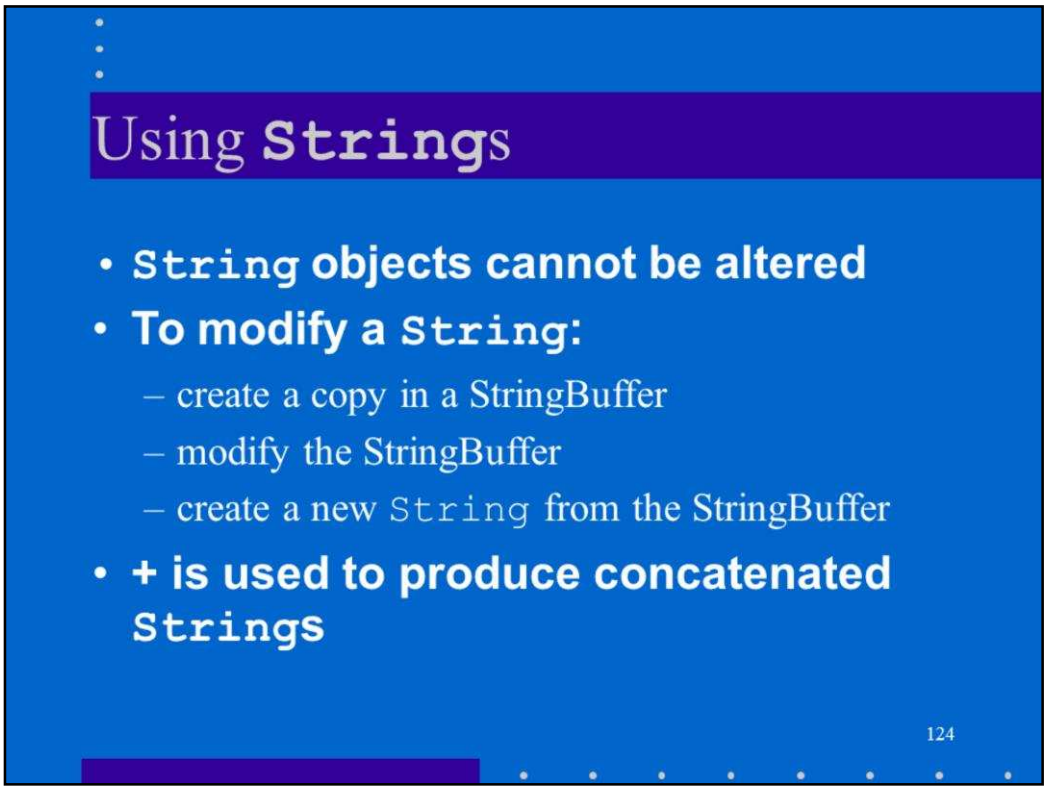
`x` is an object reference variable which refers to the `String` with value "Hi".

```
char [] c = {'H','i','p','p','o'};
String y = new String(c);
```

`y` is an object reference variable which refers to a `String` with value "Hippo".

```
char [] c = {'H','i','p','p','o'};
String z = new String(c,0,2);
```

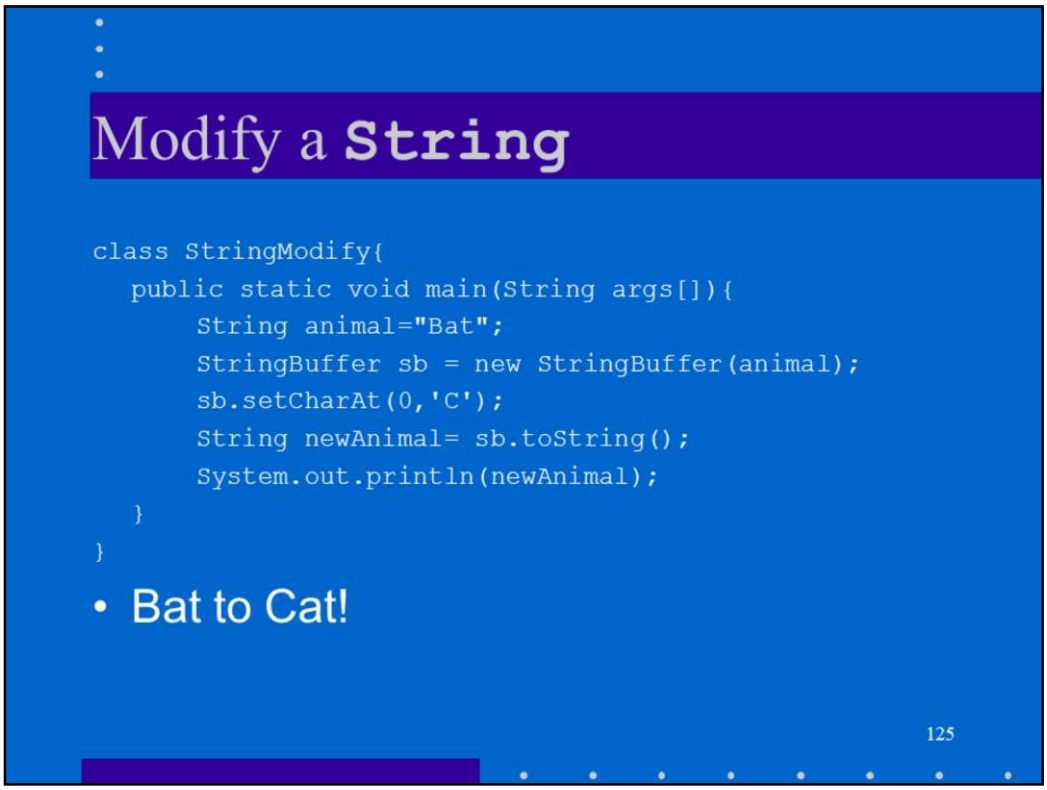
`z` is an object reference variable which refers to a `String` containing two characters of which the first can be found in `c[0]`, that is the `String` with value "Hi".


Using Strings
• String objects cannot be altered
• To modify a String:
– create a copy in a StringBuffer
– modify the StringBuffer
– create a new String from the StringBuffer
• + is used to produce concatenated Strings
124

Once created a `String` can't be altered. If you want an altered version then you have to create a new object. If after this, the original `String` is no longer referenced, then its storage will be released. The next slide shows how this is done.

Java does not implement operator overloading. So, in general classes don't have operators associated with them. String, however, is an exception, it overloads the + operator to mean string concatenation. No other Java class overloads an operator. The Java manual describes + operator in this way:

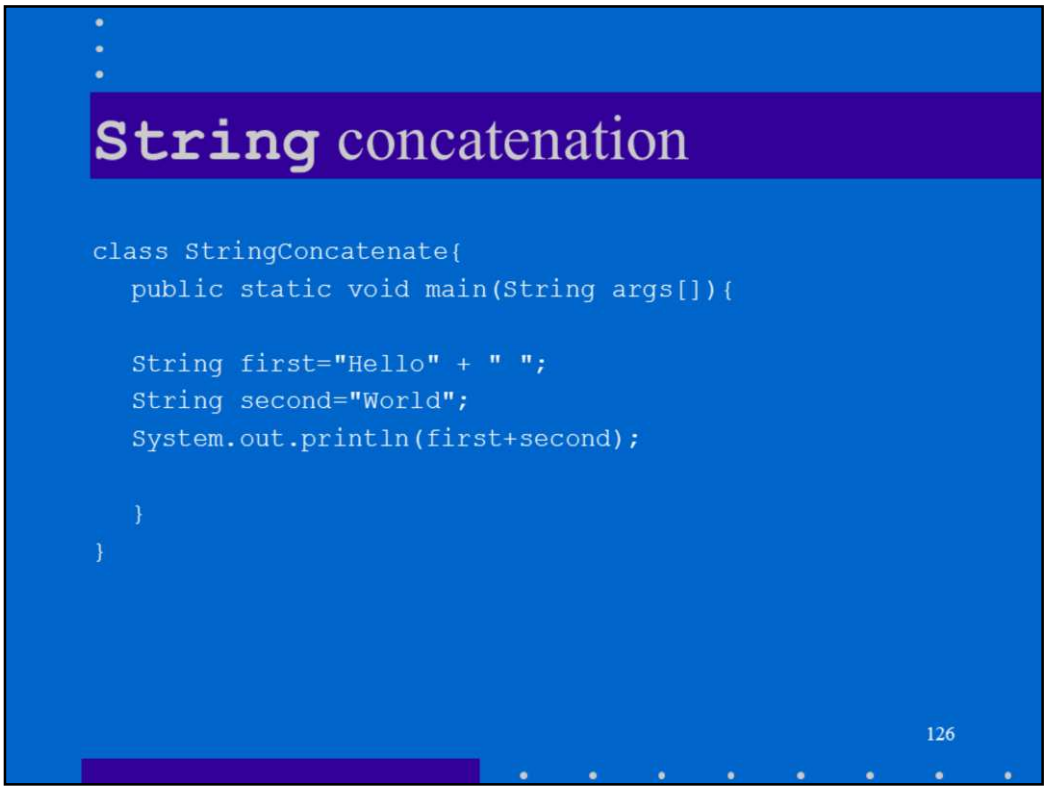
"*The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the* `StringBuffer` *class and its* `append` *method. String conversions are implemented through the method* `toString`*, defined by* `Object` *and inherited by all classes in Java*."



This code shows how a `String` can be modified to produce a new `String`. The program initialises a variable `animal` to reference a `String` containing `"Bat"`.

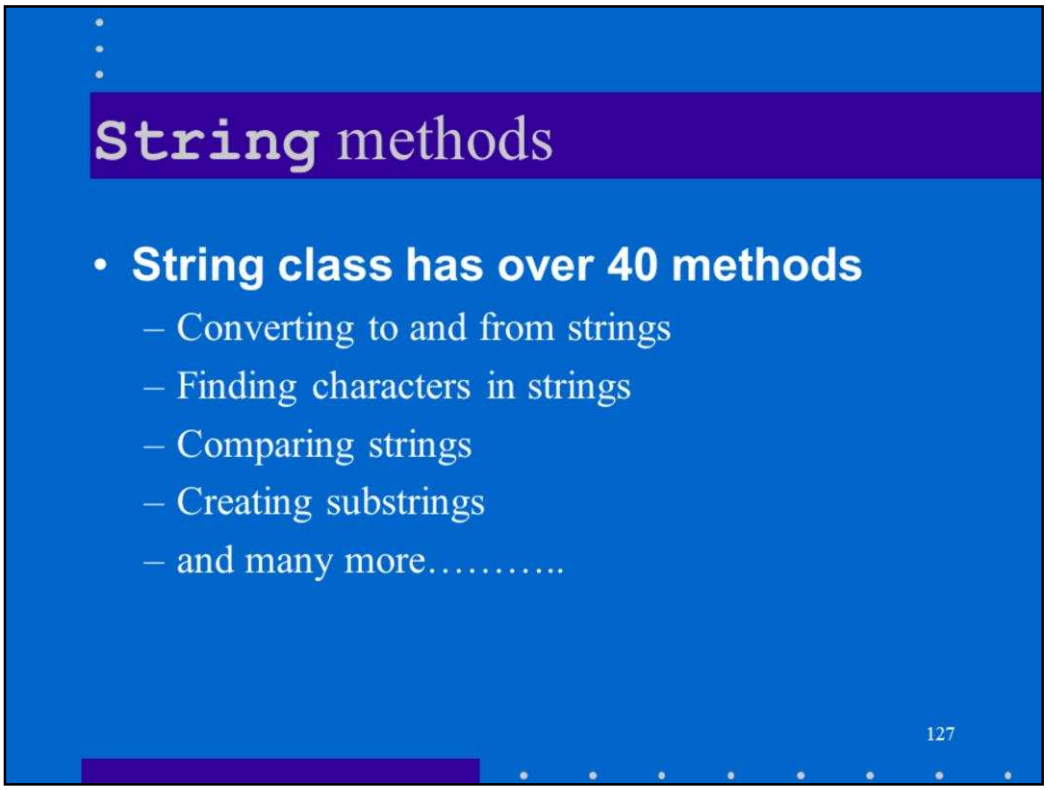
A StringBuffer is created to hold the contents of `String`. The variable `sb` is set to refer to this StringBuffer. The first character of the StringBuffer is set to `'C'` using the method `setCharAt()`. A new `String` is created and initialised with the contents of the StringBuffer referenced by `sb` which now contains `'C','a', 't'`.

N.B. We are now in a position to note that the parameter of the `main()` method (called in this case args) of an application is a `String` array. args is used to pass command line arguments to a program. The number of arguments may vary but can be obtained via the array variable args.length



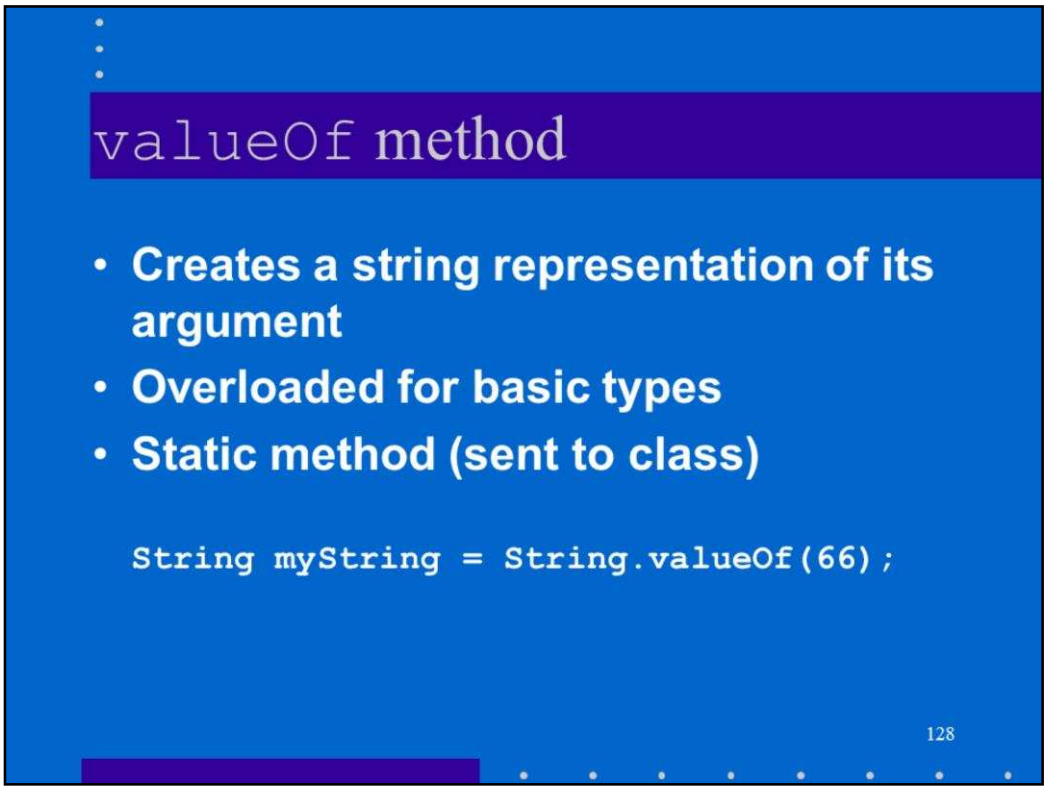
The example shows `String` concatenation. The `String` referenced by the variable named `first` is initialised to contain the concatenation of "Hello" and a `String` literal containing a space. The `String` object referenced by the variable `second` is initialised to `"World"`. The `String` referenced by `first` and the `String` referenced by `second` are concatenated and printed out. The result is:

```
Hello World
```



`String` is particularly rich in the methods it supports. A full list can be found in local Java documentation and at:
http://java.sun.com/j2se/1.3/docs/api/java/lang/String.html

Documentation of all the standard Java classes supplied with the Java Development Kit can be found at:
http://java.sun.com/j2se/1.3/docs/api/index.html


valueOf method
• Creates a string representation of its argument
• Overloaded for basic types
• Static method (sent to class)
String myString = String.valueOf(66);
128

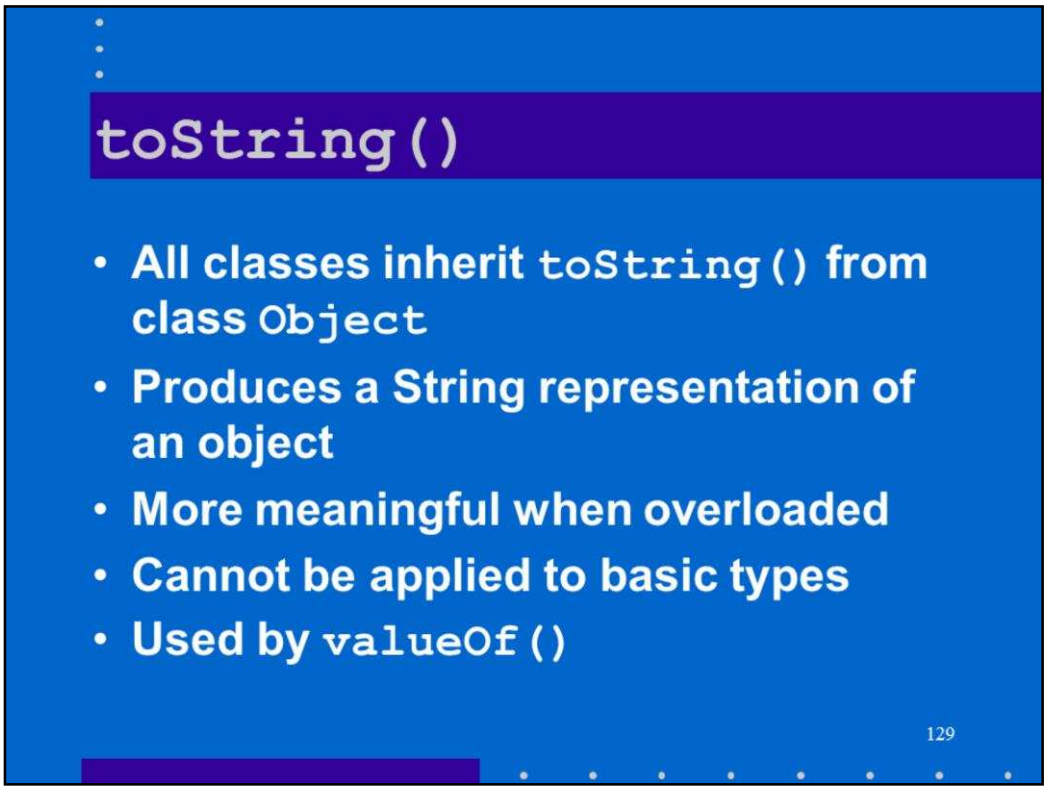
The `valueOf` method is used extensively. It accepts an argument and creates a string representation of that argument. It will work for any class but the usefulness of the output will depend on how helpful the creator of the class has been.

As we shall see, the `valueOf` method relies on the fact that all classes inherit from `Object`, so it might not necessarily work for basic Java types. Fortunately the method has been overloaded so that it accepts any basic type as well.

The method is declared as `static` and therefore it is not necessary to create a `String` object in order to invoke it.

```
String myString = String.valueOf(66);
```

will create a `String` object containing `"66"` and load the variable `myString` with a reference to the `String`.


toString()
• All classes inherit toString() from class Object
• Produces a String representation of an object
• More meaningful when overloaded
• Cannot be applied to basic types
• Used by valueOf()
129

All classes inherit from Object and one of the methods they inherit is toString(). The definition of toString() (from the JDK definiton) is:

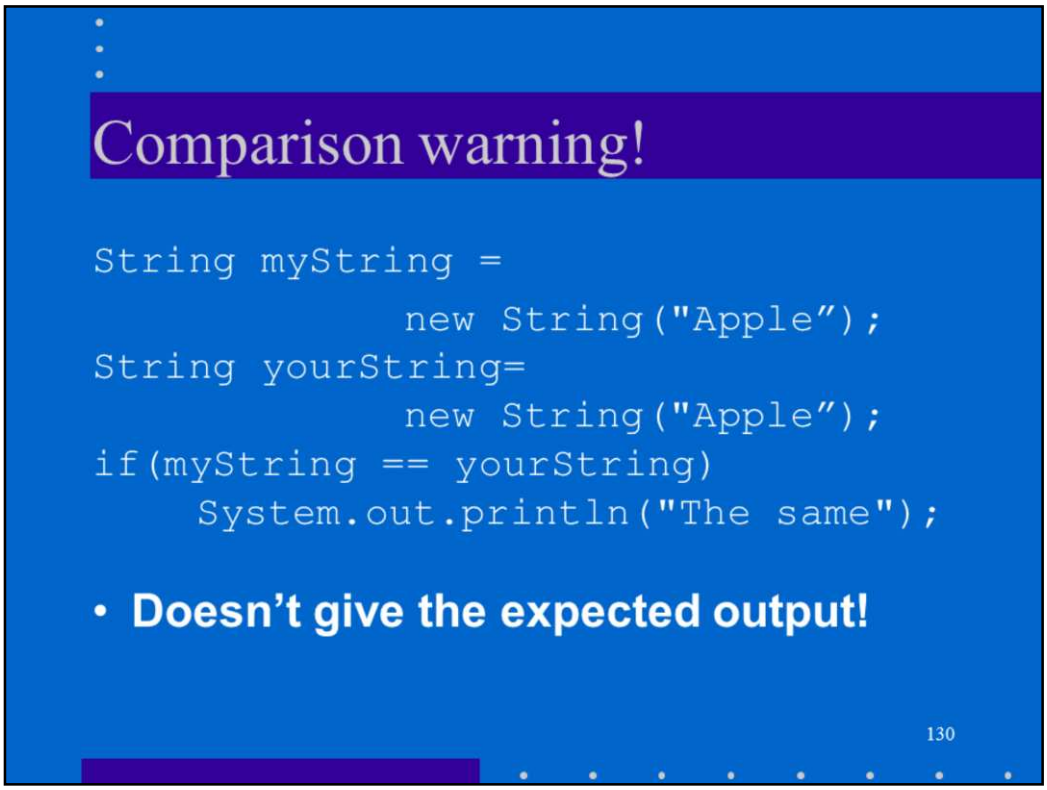
"Returns a string representation of the object. In general, the `toString` *method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.*

The `toString` *method for class* `Object` *returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object."*

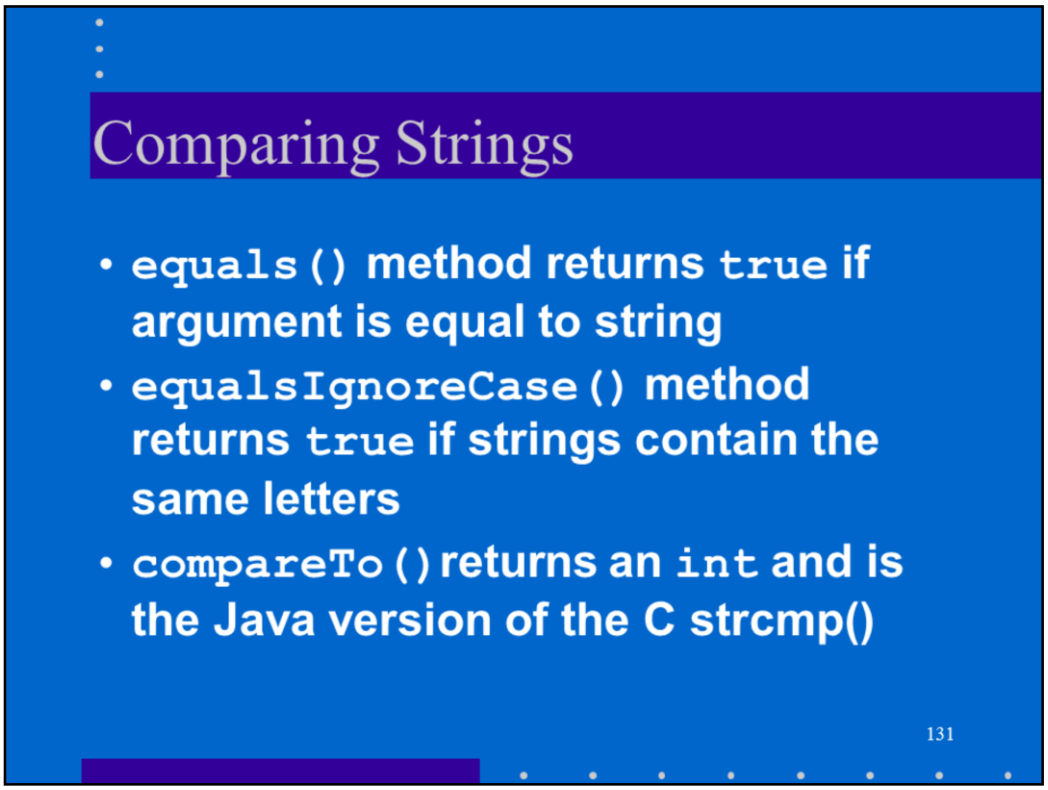
The second paragraph describes what will be output if `toString()` is not overridden. Authors of classes can produce much more helpful output by overriding the method.

Since `toString()` is a method it cannot be applied to basic types such as `int`.

`toString()` is used by `valueOf()` in order to generate a String. The value of the `String` returned by `valueOf()` depends on how `toString()` has been overloaded. `valueof()` is used in turn by `println()` to produce output.



Strings are objects not instances of basic types. Consequently, you have to be careful when you compare them. If we run this we might not get the answer we expect. Remember that `myString` and `yourString` are object references and therefore are only equal if they refer to the same object. In the example shown `myString` and `yourString` quite clearly refer to different objects and are therefore not equal!


Comparing Strings
• equals() method returns true if argument is equal to string
• equalsIgnoreCase() method returns true if strings contain the same letters
• compareTo() returns an int and is the Java version of the C strcmp()
131

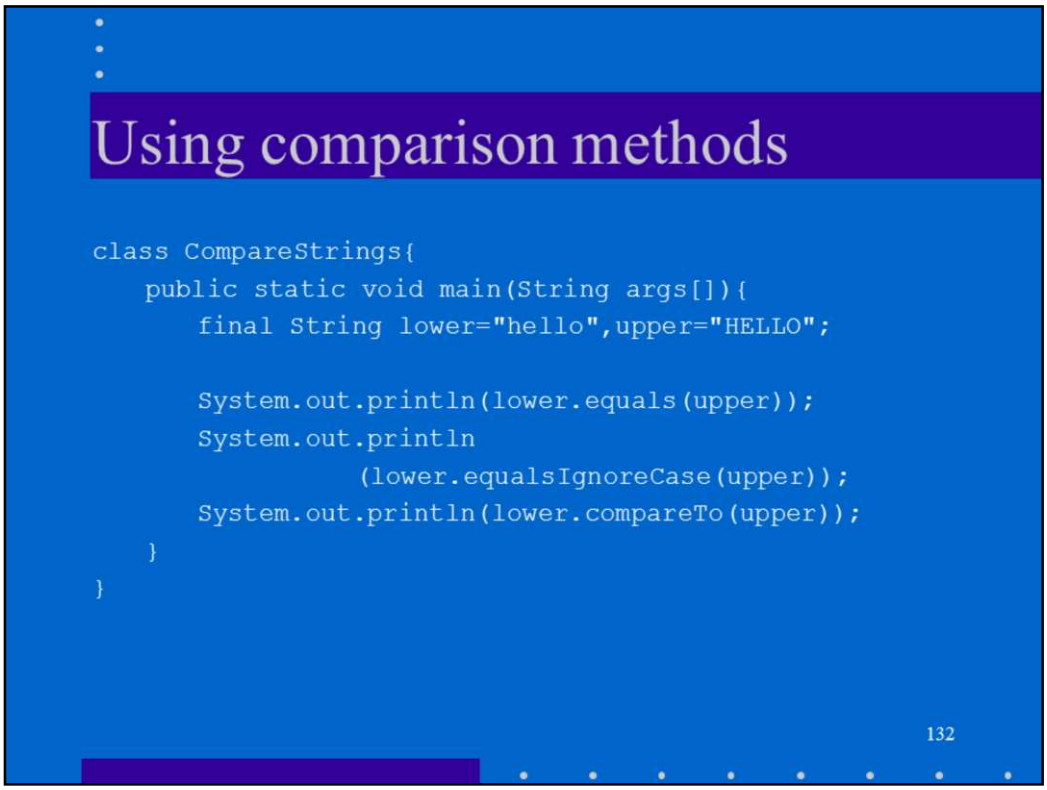
We need to do something else if we wish to test whether they refer to `String`s with the same value. `String` implements methods to do this.

`public boolean equals(anObject)`

Compares this string to the specified object. The result is `true` if and only if the argument is not null and is a `String` object that represents the same sequence of characters as this object.

`public boolean equalsIgnoreCase( anotherString)`

Compares this `String` to another object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object, where case is ignored.



This program initialises two strings. On is set to "HELLO" the other is set to "hello".

When compared using the equals() method the answer is false. The strings do not contain identical characters.

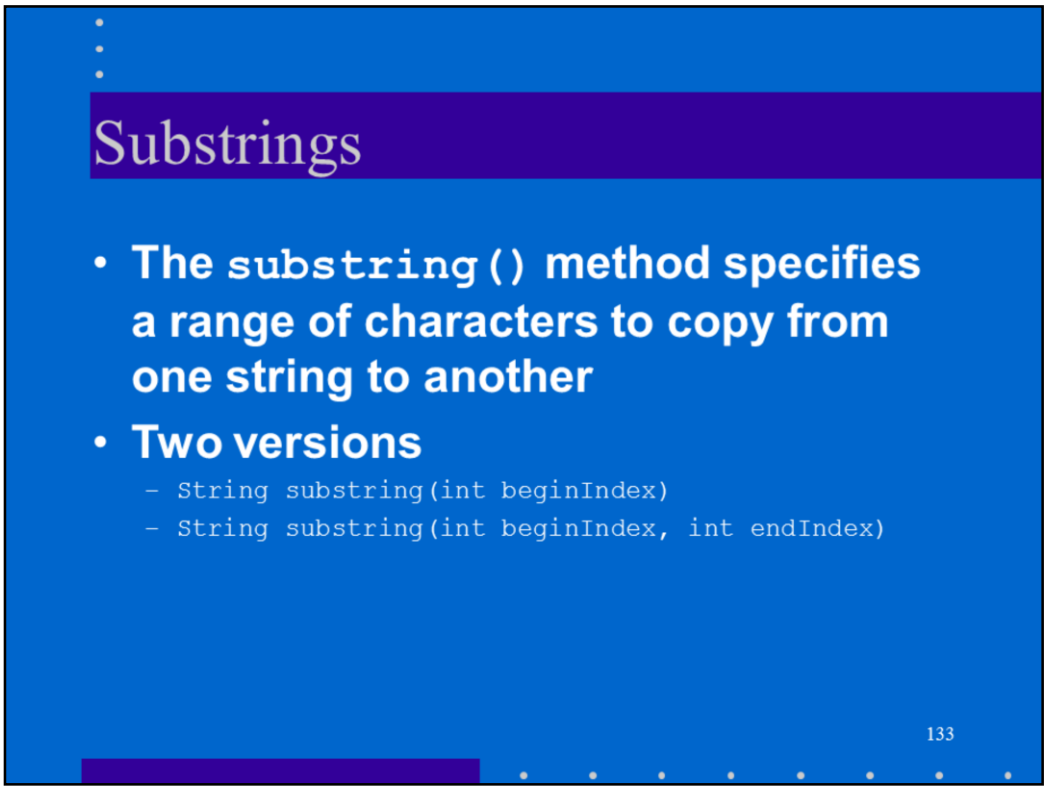
When compared using the equalsIgnoreCase() method the answer is true. The characters are the same if case is ignored.

When compared using compareTo() the answer is 32! This means that string referenced by lower is higher in the sorting sequence than the string referenced by higher.

```
public int compareTo( anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings.

Returns the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.


Substrings
• The substring() method specifies a range of characters to copy from one string to another
• Two versions
– String substring(int beginIndex)
– String substring(int beginIndex, int endIndex)
133

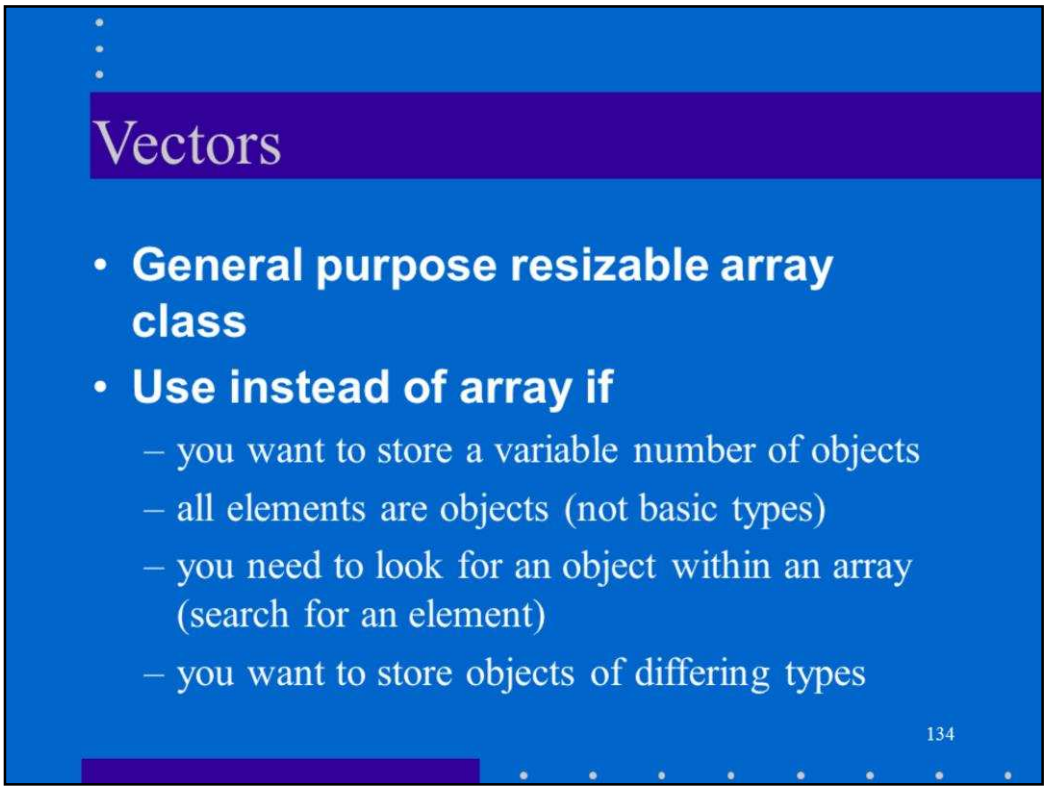
`public String substring(int beginIndex)`

Returns a new string that is a substring of this string. The substring begins at the specified index and extends to the end of this string.

`public String substring(int beginIndex,int endIndex)`

Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1.

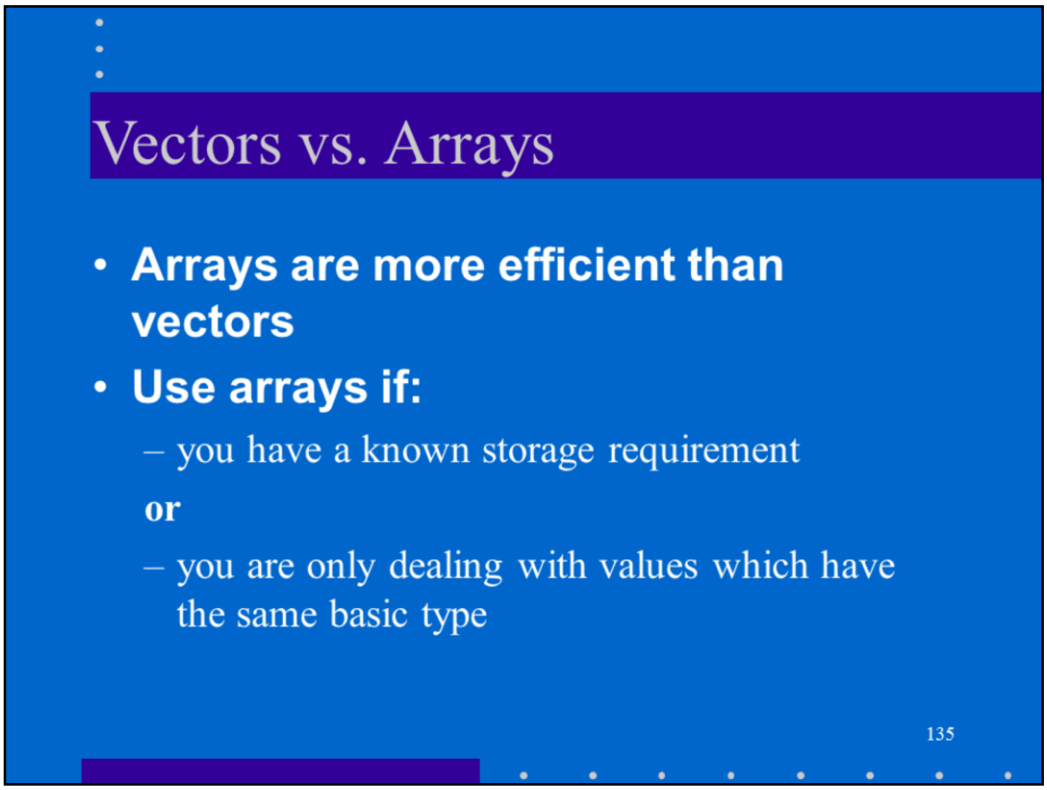
N.B.: These methods can throw a `StringIndexOutOfBoundsException` (see later for discussion of exceptions).


Vectors
• General purpose resizable array class
• Use instead of array if
– you want to store a variable number of objects
– all elements are objects (not basic types)
– you need to look for an object within an array (search for an element)
– you want to store objects of differing types
134

Arrays have a fixed size and must be assigned that size when they are created. `Vector`s act like arrays (you can pick out elements of a `Vector` using a subscript) but can grow and shrink. Java implements `Vector` as a class. Unlike array and `String` it behaves in the same way every other class does.

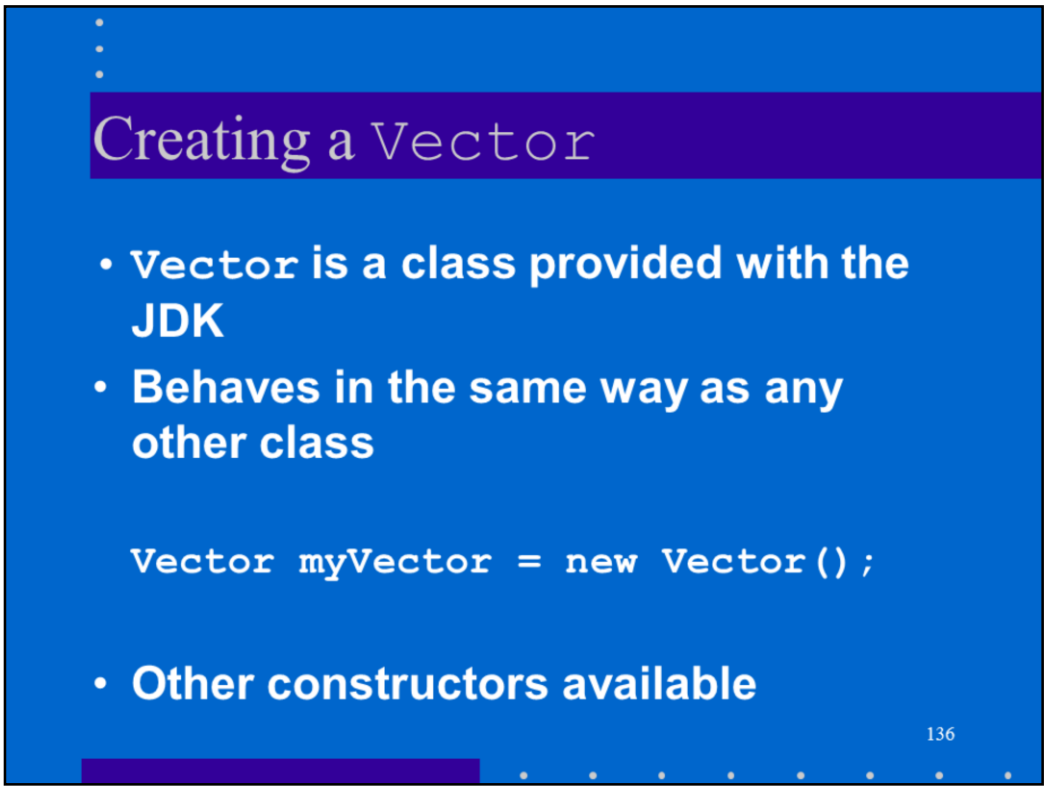
You can only store objects in a vector, variables declared as `int`, `float` etc (basic types) can't be stored in a vector. If you want to create a `Vector` which contains integers then you need to use a wrapper class (`Integer`, `Float` etc.).

`Vector`s implement methods for searching. The obvious way to search an array is to start at the beginning and work through each array member in turn until the array member you want is found. `Vector`s implement much more efficient searching mechanisms.


Vectors vs. Arrays
• Arrays are more efficient than vectors
• Use arrays if:
– you have a known storage requirement
or
– you are only dealing with values which have the same basic type
135

It is costly to have a structure which can grow in size. Also `Vectors` can only be accessed via methods whereas arrays can be accessed by adding an offset to the start address of the array. Its is therefore much quicker to access an element in an array than an element in a `Vector`.

So if you know the maximum number of elements you will ever want to store then an array may be more efficient than a `Vector`.


Creating a Vector
• Vector is a class provided with the JDK
• Behaves in the same way as any other class
Vector myVector = new Vector();
• Other constructors available
136

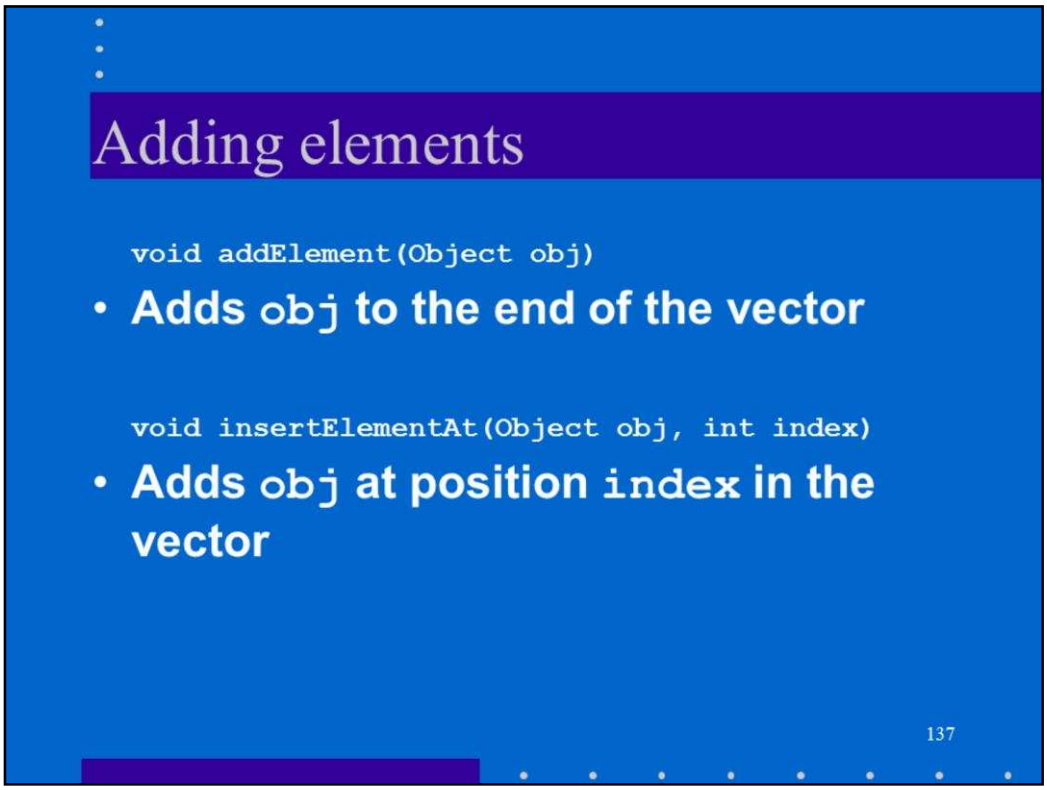
A number of classes are provided with Sun's Java Development Kit (JDK). These are grouped into packages (see next weeks lecture). Using a package, normally means that it must be referenced with the `import` statement. For some classes this is not necessary as they belong to the `java.lang` package which is available to every other class. `Vector`, however, belongs to the package called `java.util`.

`Vector` behaves in the same way as a user created class. It implements methods and constructors and no special operators are associated with it.

`Vector`s are created in the same way that any other object is created i.e via the `new` operator.

Vector has three constructors:

`public Vector(int initialCapacity,int capacityIncrement)`
Constructs an empty vector with the specified initial capacity and capacity increment.
`public Vector(int initialCapacity)`
Constructs an empty vector with the specified initial capacity.
`public Vector()`
Constructs an empty vector.


Adding elements
void addElement(Object obj)
• Adds obj to the end of the vector
void insertElementAt(Object obj, int index)
• Adds obj at position index in the vector
137

```
public final synchronized void addElement(Object
obj)
```

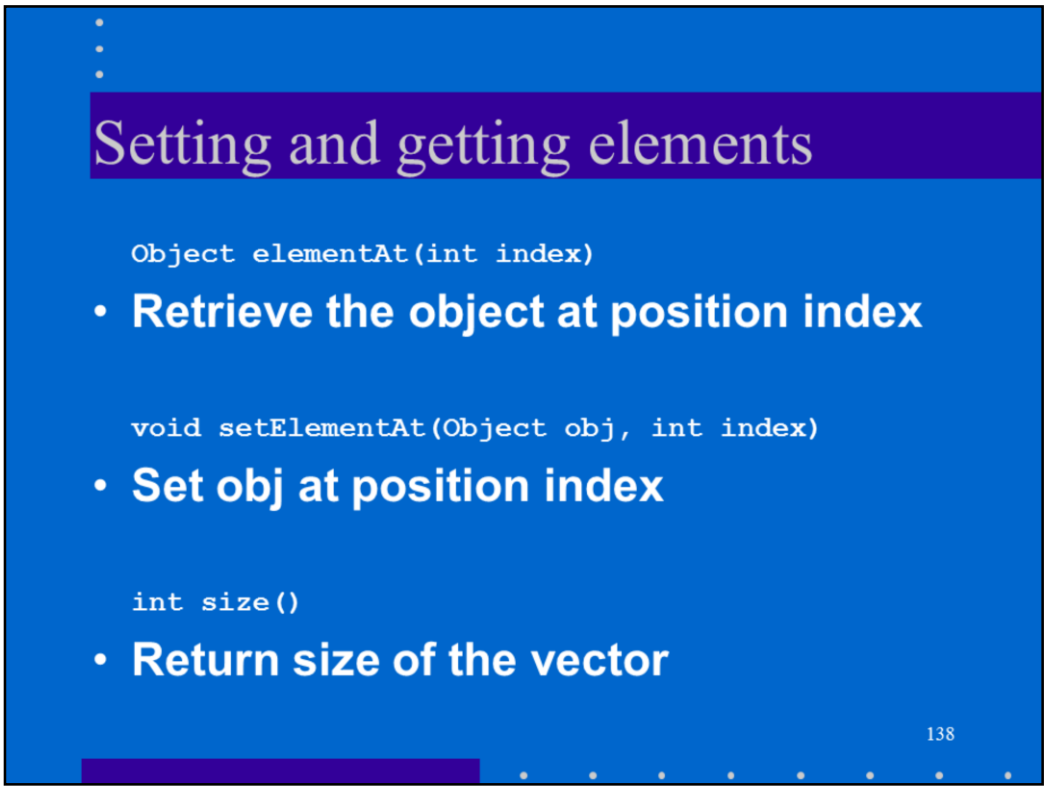
Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

```
public final synchronized void insertElementAt(
Object obj,int index)
```

Inserts the specified object as a component in this vector at the specified index. Each component in this vector with an index greater or equal to the specified index is shifted upward to have an index one greater than the value it had previously.

The index must be a value greater than or equal to 0 and less than or equal to the current size of the vector.

The difference between size and capacity is that size is a measure of the current number of elements in a vector, capacity is a measure of the number of object references which may be added to the vector before the Java runtime system needs to allocate more memory to it.



`public final synchronized Object elementAt(int index)`

Returns the component at the specified index.

`public final synchronized void setElementAt( Object obj,int index)`

Sets the component at the specified index of this vector to be the specified object. The previous component at that position is discarded.

The index must be a value greater than or equal to 0 and less than the current size of the vector.

`public final int size()`

Returns the number of components in this vector.

The full details of the methods implemented in the Vector class can be found at:

http://java.sun.com/j2se/1.3/docs/api/java/util/Vector.html

Vector example

```
import java.util.*;
class VectorTest{
   public static void main(String args[]){
        Vector hector = new Vector();

        hector.addElement("apple");
        hector.addElement("banana");
        hector.addElement("date");
        for(int i=0;i<hector.size();i++)
              System.out.println(hector.elementAt(i));
   }
}
```

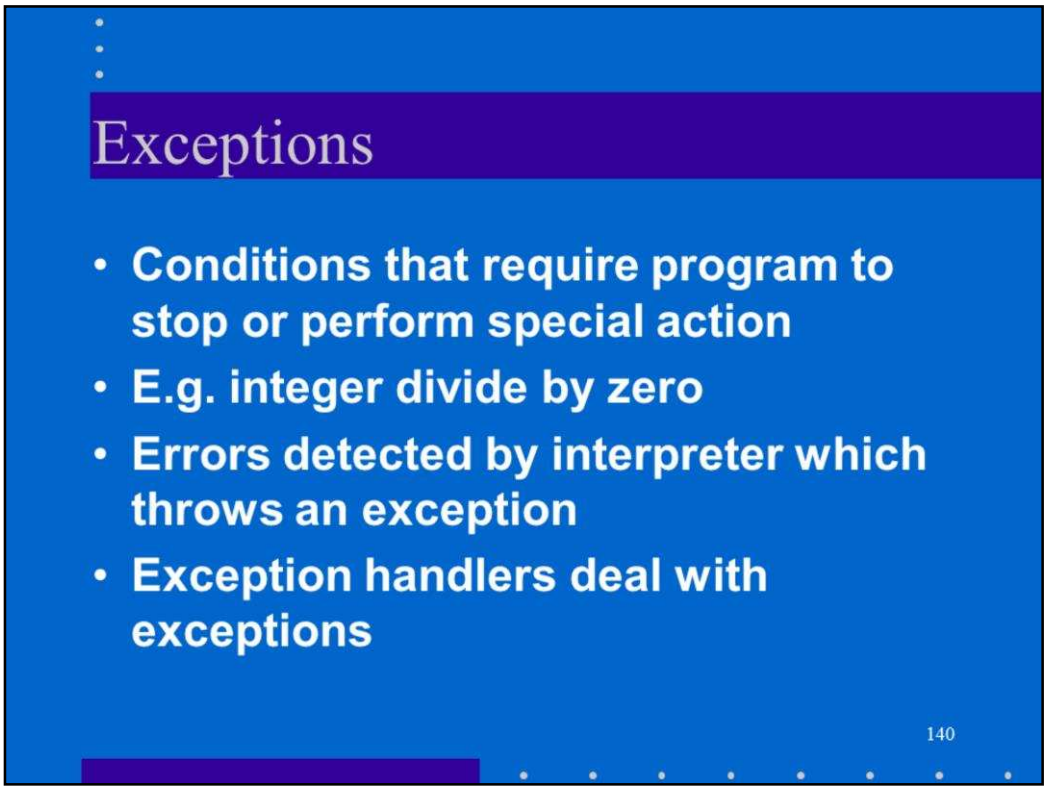
139

In this program an object reference variable called `hector` references a `Vector`. The `Vector` is not given a length. Three `String`s are added to the `Vector`. These will go into position `0`, `1` and `2`.

A loop prints out elements `0`,`1` and `2` of the `Vector`.

Note the `import` statement for `java.util`.

Since `Vector`s store `Object`s, it is normally necessary to cast the result of applying `elementAt` to the correct type.


Exceptions
• Conditions that require program to stop or perform special action
• E.g. integer divide by zero
• Errors detected by interpreter which throws an exception
• Exception handlers deal with exceptions
140

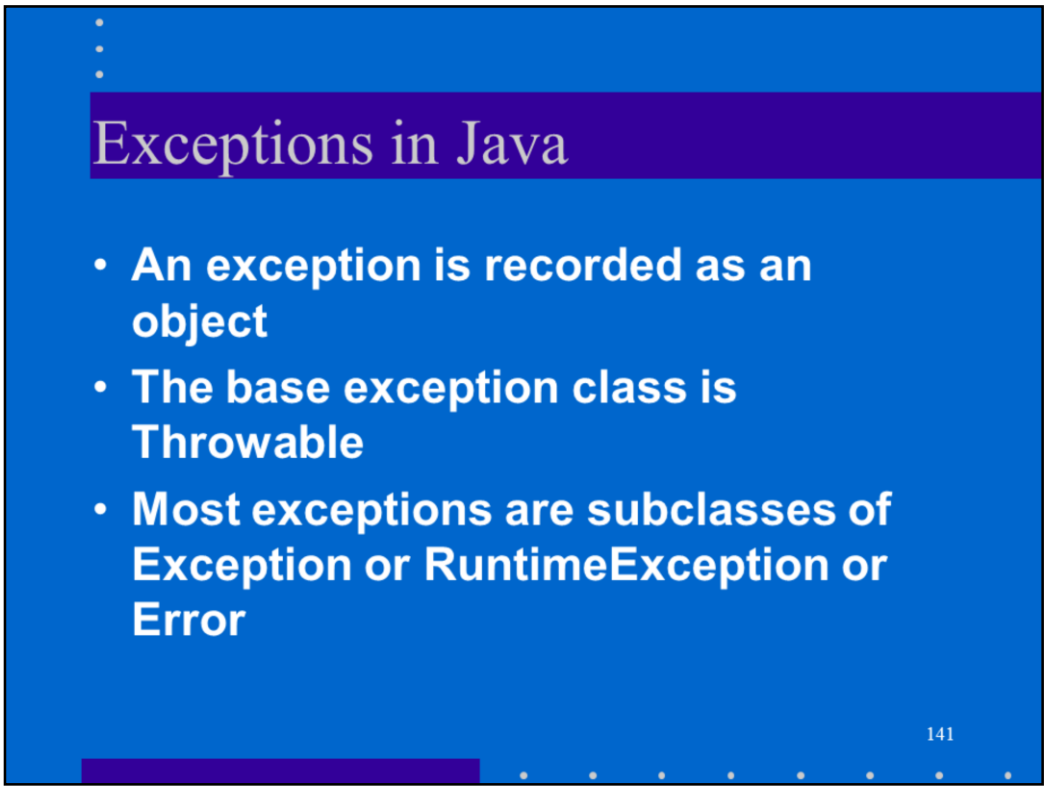
An exception is something that may occur during the execution of a program. It isn't entirely unexpected but its something which normally shouldn't occur.

For example, suppose a program contains a loop which continually divides an integer x by another integer y. On occasions y may be set to zero and the result of the division is undefined.

In this situation some programming languages would simply report an error to the operating system and the program would fail. Java, however, raises an exception. If the programmer has chosen to deal with the type of exception that has been raised then control will pass to the code assigned to this exception, otherwise, as with other languages, the program will exit to the operating system.

Java allows programmers to write exception handlers to deal with situations that might arise during the execution of a program. For example, in a routine to read numbers from the keyboard it is possible to write an exception handler which will sort out problems caused by a user typing an alphabetic character when a numeric character is required.

The Java Specification defines an exception in the following way: "When a Java program violates the semantic constraints of the Java language, a Java Virtual Machine signals this error to the program as an *exception*. An example of such a violation is an attempt to index outside the bounds of an array."


Exceptions in Java
• An exception is recorded as an object
• The base exception class is Throwable
• Most exceptions are subclasses of Exception or RuntimeException or Error
141

"Every exception is represented by an instance of the class `Throwable` or one of its subclasses; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it."

The classes `Exception` and `Error` are subclasses of `Throwable`. `RuntimeException` is a subclass of `Exception`. `Exception` is the superclass of all exceptions that ordinary programs can be expected to recover from. `Error` is the superclass of all exceptions from which ordinary programs are not expected to recover, for example, internal errors in the Java Virtual Machine.

Exceptions may be *checked* or *unchecked*. The Java compiler will flag an error if a program does not supply an exception handler for all *checked* exceptions that might occur. Examples of checked exceptions are `ClassNotFoundException`, `IllegalAccessException`, `IOException`.

`RuntimeException` is the superclass of all unchecked exceptions. Examples of runtime exceptions are `ArithmeticException` and `ArrayStoreException`. The standard exceptions are contained in the package java.lang and are documented at http://java.sun.com/j2se/1.4/docs/api/java/lang/package-frame.html

Programmers may define their own exception classes.

Exception example

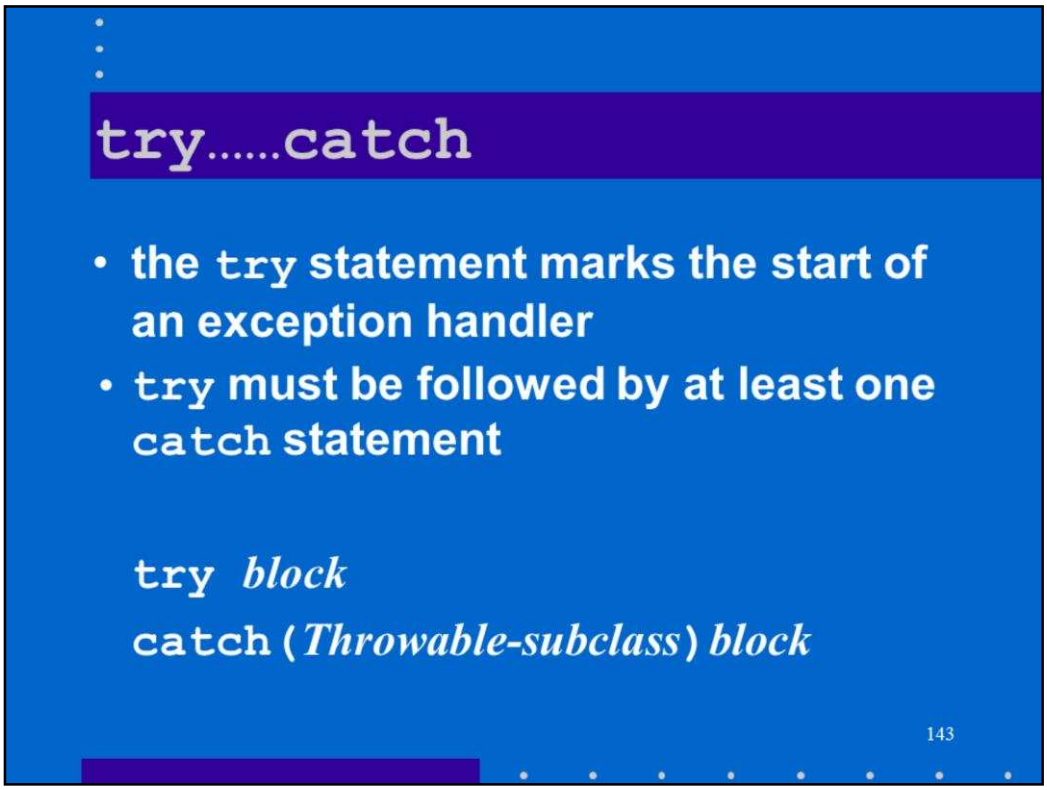
```
class ExceptionExample{
   public static void main(String args[]){
        int[] myArray = new int[3];
        try{
              for(int i=0;i<4;i++) {
                    myArray[i]=0;
              }
        }catch(Exception e){
              System.out.println("Exception!");
        }
        System.out.println("End of program");
   }
}
```

142

The array `myArray` is created to hold three integers. The `for` loop has been incorrectly coded so that it will try to access `myArray[3]` (beyond the last element). Normally this would throw an `ArrayIndexOutOfBoundsException` which would cause the program to halt immediately.

In this case the Java exception handling mechanism has been used to detect the exception. The exception is still generated but execution of the program will continue outside of the loop at fault.

The array will be filled, then on the fourth iteration of the loop an exception will be raised. In this case all the exception handler does is print out a message to say that an exception has occurred. The program then prints a final message to say that it has ended. This message is part of the normal operation of the program and would appear if the loop were amended to avoid generating the exception.


try……catch
• the try statement marks the start of an exception handler
• try must be followed by at least one catch statement
try block
catch (Throwable-subclass) block
143

Exception handlers have two parts. a `try` block with associated catch blocks. The `try` statement indicates which statements are to be dealt with by the exception handler. Only exceptions which are generated from statements within that block are handled by the `catch` statements.

`catch` statements specify what will happen when an exception occurs. Since exceptions from a number of exception classes could occur in any piece of code there can be a number of `catch` statements associated with a try block, each handling a different exception class.

A `catch` statement specifies what class of exception it is handling and the code that should be executed when an exception belongs to the class occurs.

`catch (Exception e){……}` specifies a handler for all the exceptions that can be reasonably handled by a program.

It is often necessary to be more specific, for example:

```
catch (NullPointerException e){……..}
```

A `NullPointerException` could potentially be handled by either of these `catch` statements because `NullPointerException` is a subclass of `Exception`. In practice when an exception is thrown the `catch` statements are examined in the order they appear in the program and the first one which matches the class of the exception is executed.

`catch` can be regarded almost as though it were a method with a parameter of type `Exception`

Catching a given exception

```
class GivenExceptionExample{
   public static void main(String args[]){
      int[] myArray = new int[3];
      try{
         for(int i=0;i<4;i++) {
            myArray[i]=0;
         }
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Array Exception!");
      }catch(Exception e){
         System.out.println("Unexpected Exception!");
      }
      System.out.println("End of program");
   }
}
```

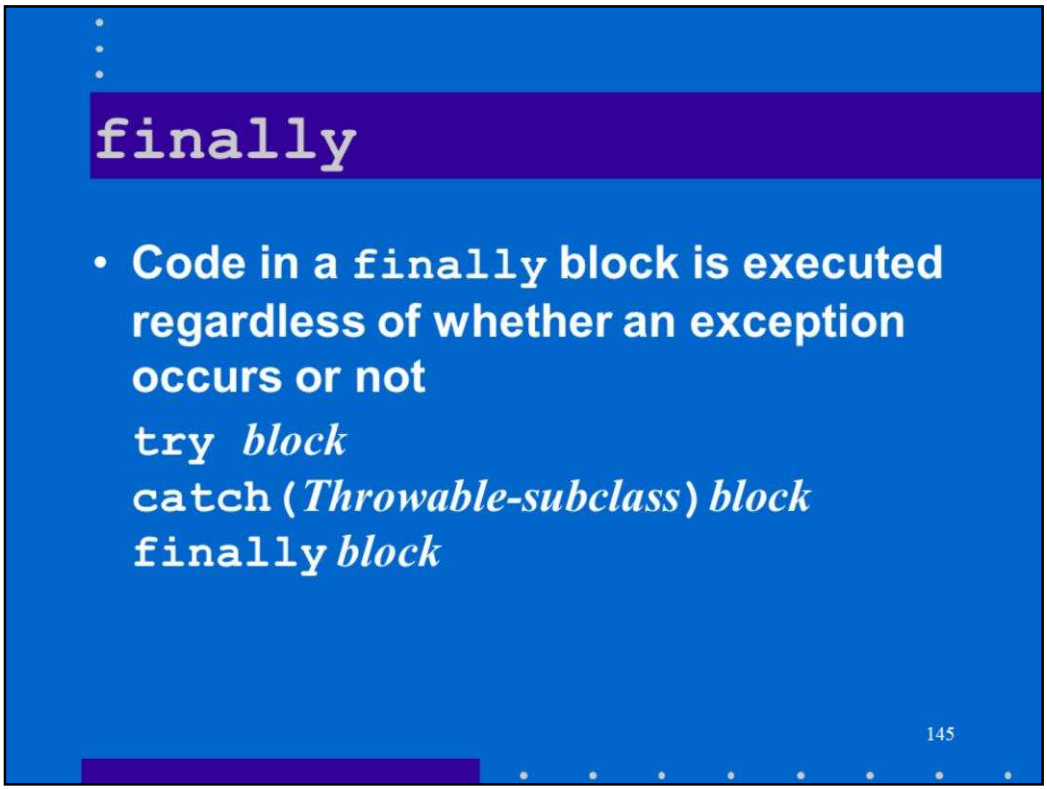
144

This example is very similar to the previous one except in this the exception handler explicitly catches the exception raised when an attempt is made to write over the end of an array.

When the program attempts to write myArray[3] an `ArrayIndexOutOfBounds` exception will be raised. This will be trapped by the first `catch` statement and the program will output "Array Exception". If any other exception occurs then the program will output "Unexpected Exception".

The program will always close by outputting "End of program".

The order in which the catch statements appear in the program is important.


finally
• Code in a finally block is executed regardless of whether an exception occurs or not
try block
catch (Throwable-subclass) block
finally block
145

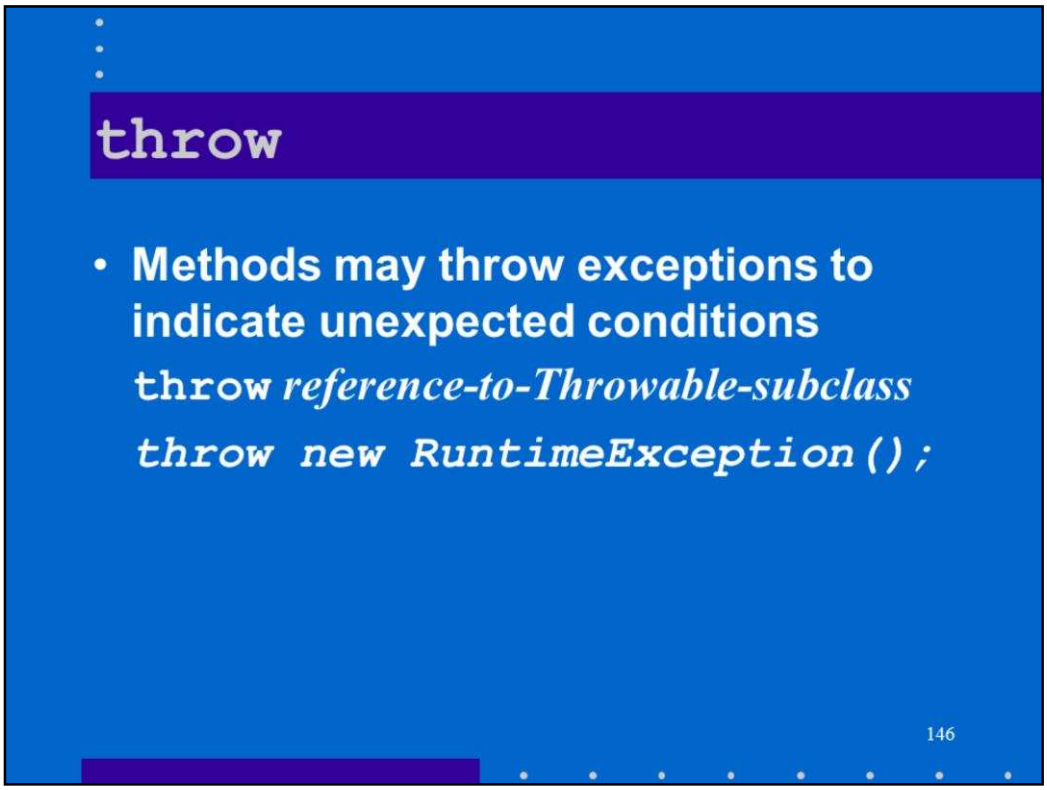
Code in a `finally` block is executed after code in a `try` block has been executed, regardless of whether an exception occurs or not.

If no exception occurs then the flow of execution leaves the `try` block and the `finally` block executes.

If an exception occurs then the appropriate `catch` block is executed and then the `finally` is executed.

Code in the `finally` block is normally written to 'tidy up' the effects of the code in the `try` block.

One of the complications of exception handling is that exceptions may be thrown in a `catch` or `finally` block!



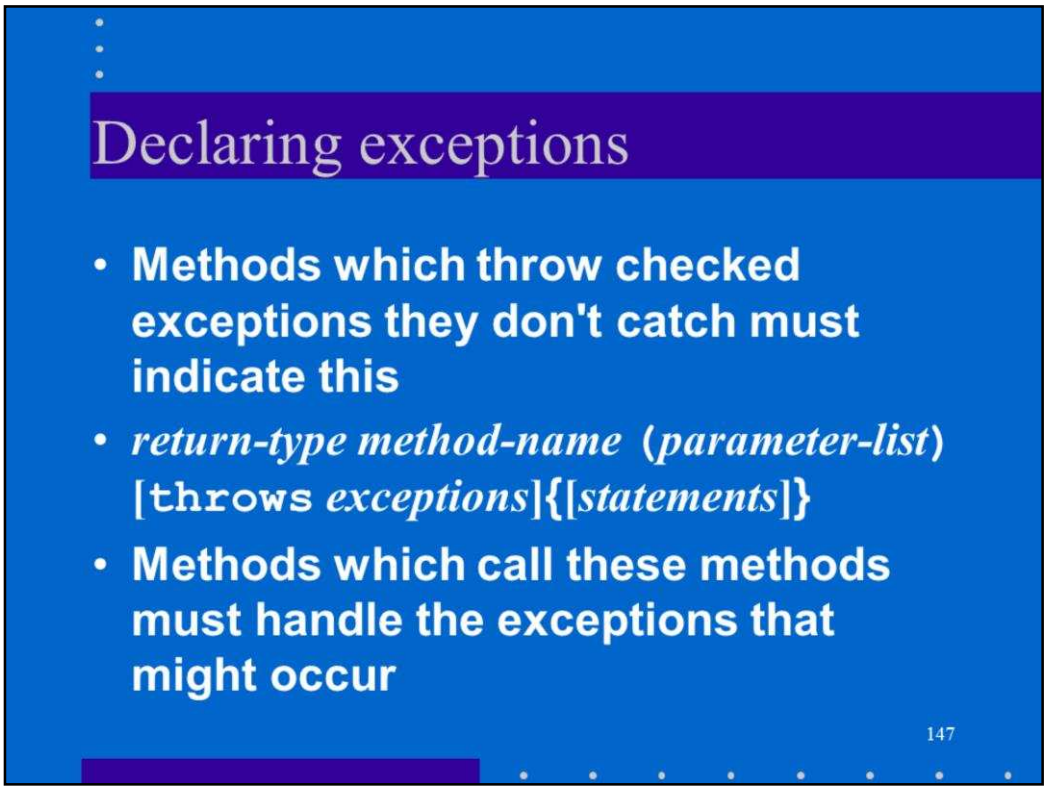
Programmers may decide to write code which will throw exceptions. Typically this is done when a routine detects something which shouldn't happen and where it is the calling routines responsibility to decide what to do next.

Suppose you were writing a library routine which expected an integer in the range 1 to 10 to be passed to it. The actual argument is tested and it is found to be outside the required range. One way of dealing this might be to pop up a warning window and exit the routine. There is no guarantee that the routine will be called by something designed to run in a windowed environment. A far safer way of highlighting the problem is to throw an exception.

This is achieved by using the `throw` statement.

The calling routine can catch the thrown exception and take appropriate action. The Java specification says:

"*A `throw` statement causes an exception to be thrown. The result is an immediate transfer of control that may exit multiple statements and multiple constructor, static and field initializer evaluations, and method invocations until a `try` statement is found that catches the thrown value. If no such `try` statement is found, then execution of the code that executed the `throw` is terminated after invocation of the `UncaughtException` method.*"


Declaring exceptions
• Methods which throw checked exceptions they don't catch must indicate this
• return-type method-name (parameter-list) [throws exceptions]{[statements]}
• Methods which call these methods must handle the exceptions that might occur
147

If you write a method which might throw a checked exception you have to indicate this in the method declaration.

Examples of checked exception classes are:

`IOException`: A requested I/O operation could not be completed normally. Subclasses of this class include:

`EOFException`: End of file has been encountered before normal completion of an input operation.

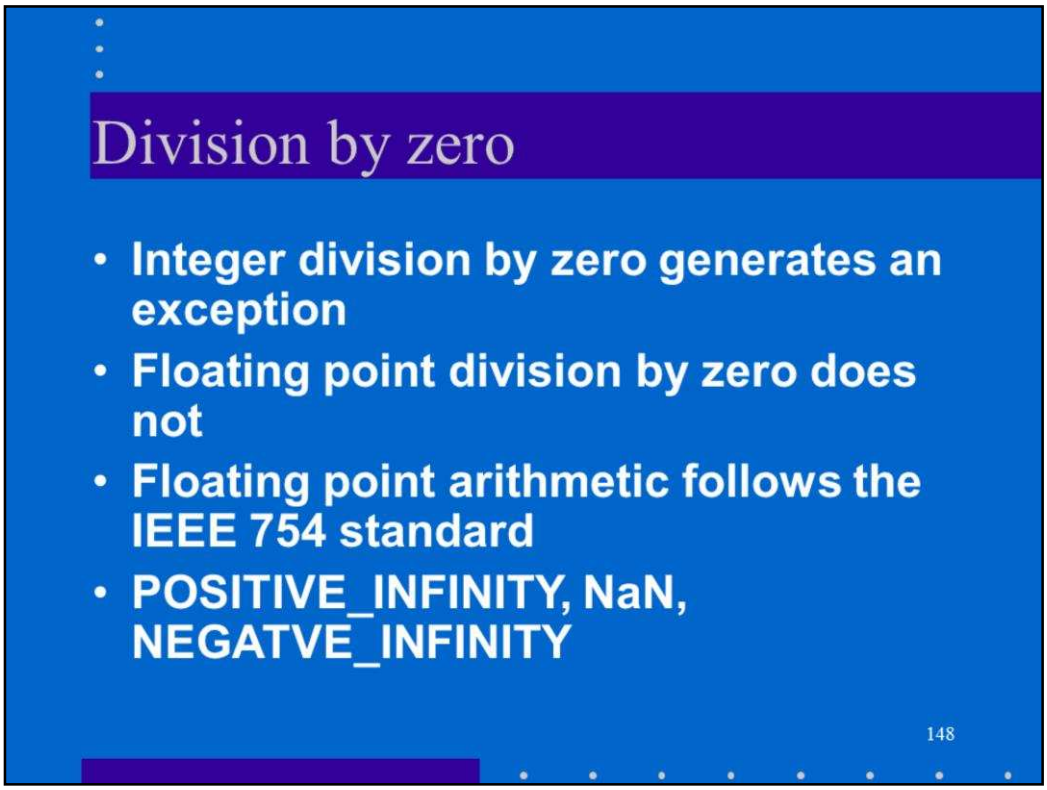
`FileNotFoundException`: A file with the name specified by a file name string or path was not found within the file system.

`InterruptedIOException`: The current thread was waiting for completion of an I/O operation, and another thread has interrupted the current thread, using the interrupt method of class `Thread`.

Methods which might throw these type of exception should indicate this in the throws clause of their declaration:

```
void myMethod() throws IOException{……..}
```

Methods calling `myMethod()` must provide an exception handler to deal with an `IOException` or else indicate in their throws clause that they may throw an `IOException`.


Division by zero
• Integer division by zero generates an exception
• Floating point division by zero does not
• Floating point arithmetic follows the IEEE 754 standard
• POSITIVE_INFINITY, NaN, NEGATVE_INFINITY
148

Suppose you have a Java language program in which the following expression appears:

```
1.0/0.0
```

What happens? Does this usage cause an exception to be thrown? Is the result undefined?

In the Java programming language, integral division by zero results in an `ArithmeticException`. But for floating-point, no exception is thrown (in C++ the result of division by zero is undefined). The result of `1.0/0.0` is positive infinity, indicated by the constant `Double.POSITIVE_INFINITY`

This example illustrates an important point, which is that Java language floating-point arithmetic operates according to a well-defined standard, known as IEEE 754.

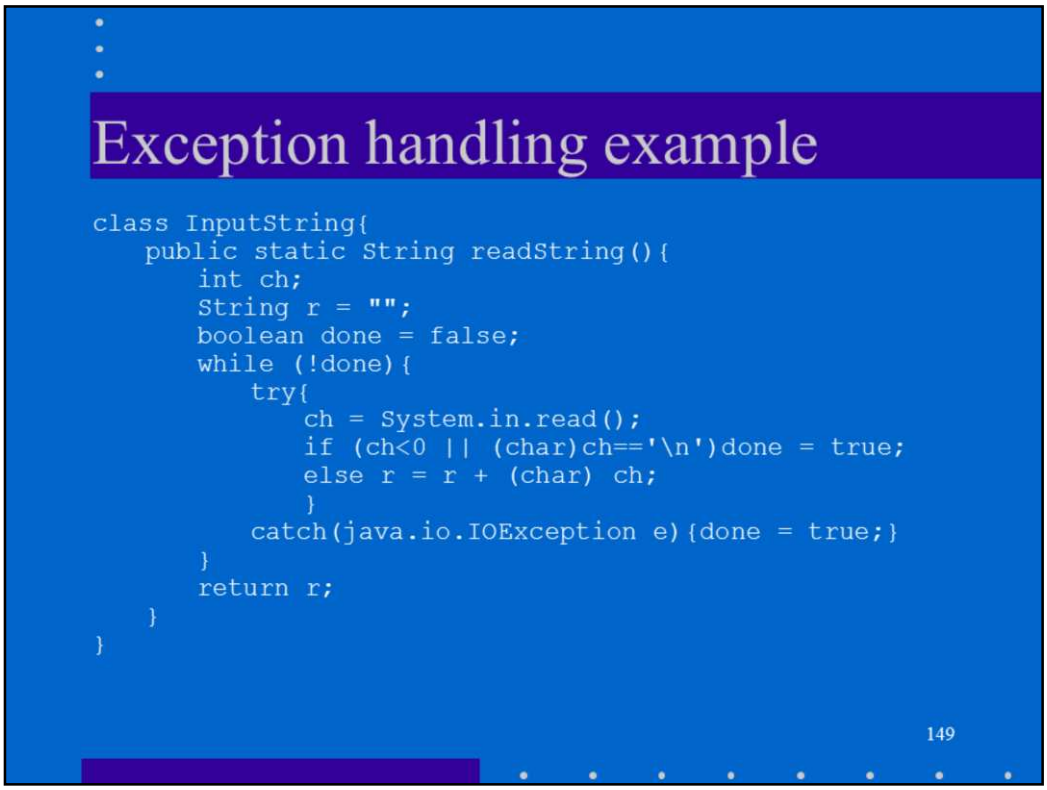
Another related idea is that of `NaN` (not a number), used to represent the results of certain arithmetic operations such as the following:

```
0.0/0.0
```

`NaN` is interesting in that it has the following property:

```
NaN != NaN
```

In other words, `NaN` is unequal to itself, and this fact is used to implement methods such as `Double.isNaN`.



The reason that, so far in the lectures, there hasn't been an example of reading a value into a program is that all standard Java input methods may throw an `IOException`. This is a checked exception and has to have an exception handler defined for it.

In this class a string is read in character by character until a new line character is detected.

The variable `System.in` references an `InputStream`. `InputStream` objects implement the method `read()`.

```
public abstract int read() throws IOException
```

`read()` returns an int containing a character from the input stream. If there are no more characters to read it returns `-1`.

The program loops round reading character after character until it encounters a new line character or the end of the stream is reached (-1 is returned). Each character is added to the string `r` which is returned to the calling routine.

If an `IOException` occurs this is taken to mean that no more characters can be read so the loop is terminated by setting the loop control variable in the catch block and the method returns the value of `r`.

If the `try….catch` blocks were not present then this code would not compile.

Reading a string

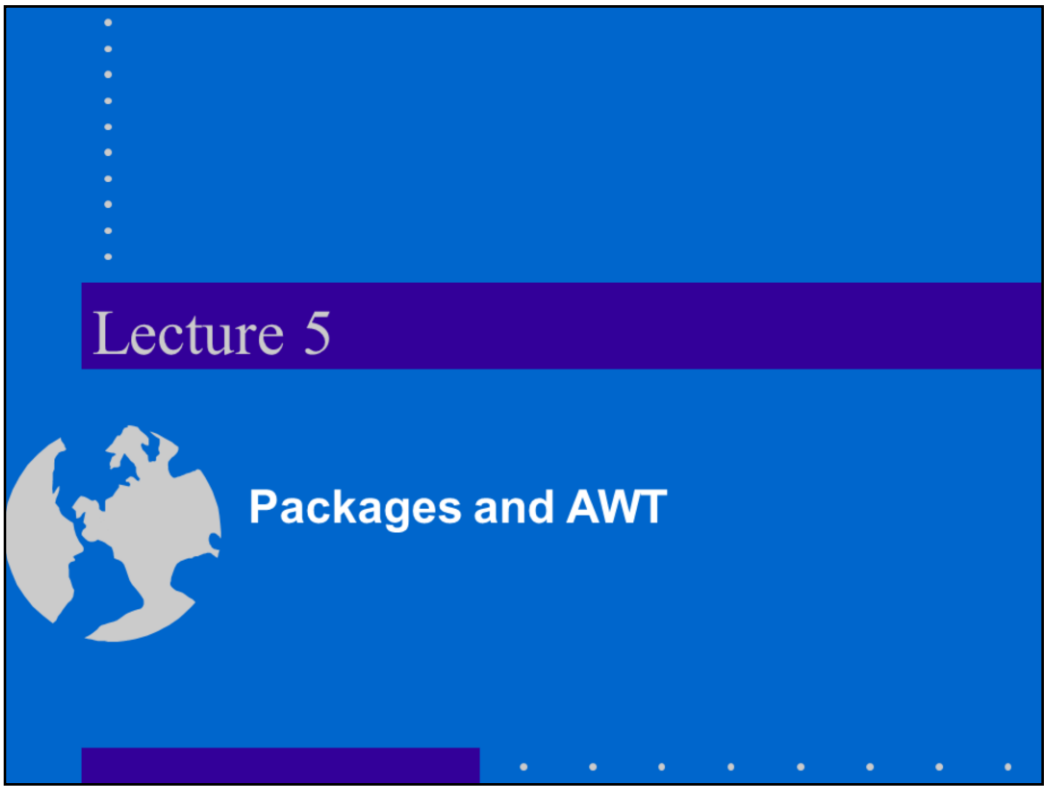
```
class TestReadString{
    public static void main(String args[]){
        String in;
        in = new String(InputString.readString());
        System.out.println(in);
    }
}
```

150

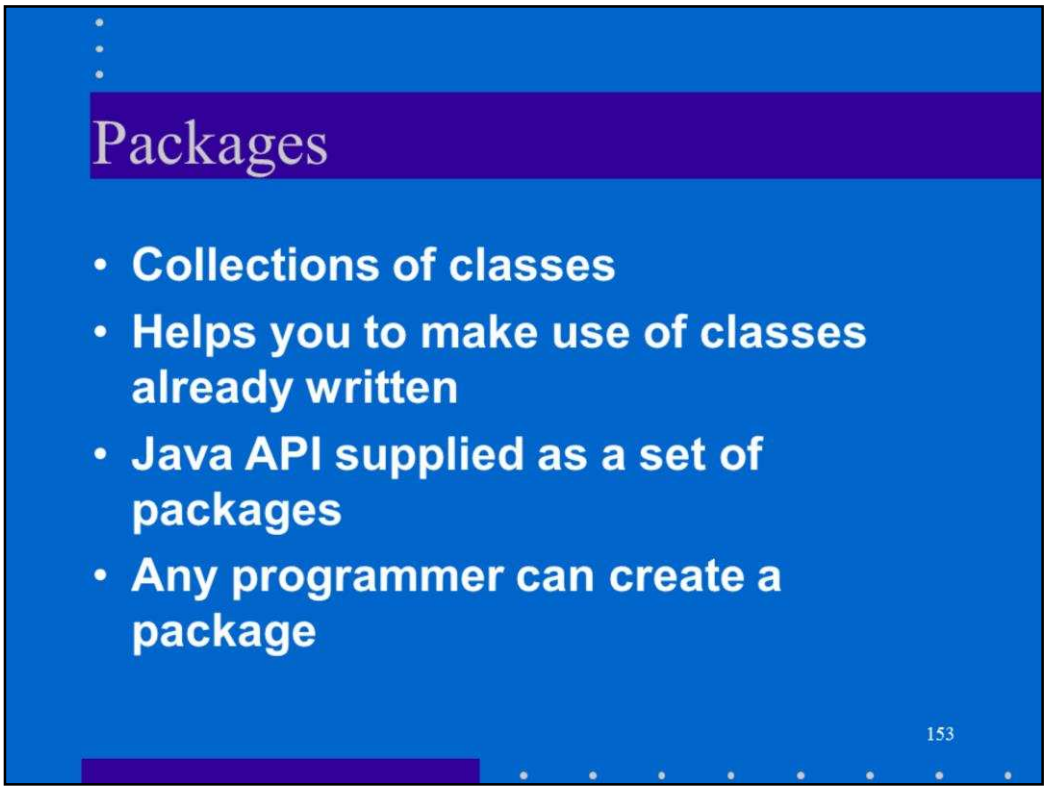
This short application calls the `readString()` method of class `InputString` (shown on the previous slide). This class does not have to handle `IOException`s because that is undertaken by `readString()`. The `String` is read into the object referenced by `in` and then printed out.

Summary

- Wrappers
- Arrays
- Strings
- Vectors
- Exceptions
- Input



Java classes may be arranged into collections known as packages. Packages are class libraries. Almost all applications and applets will make use of packages provided either as part of the JDK or by another programmer. AWT is the standard package which provides classes that can be used to construct graphical user interfaces.


Packages
• Collections of classes
• Helps you to make use of classes already written
• Java API supplied as a set of packages
• Any programmer can create a package
153

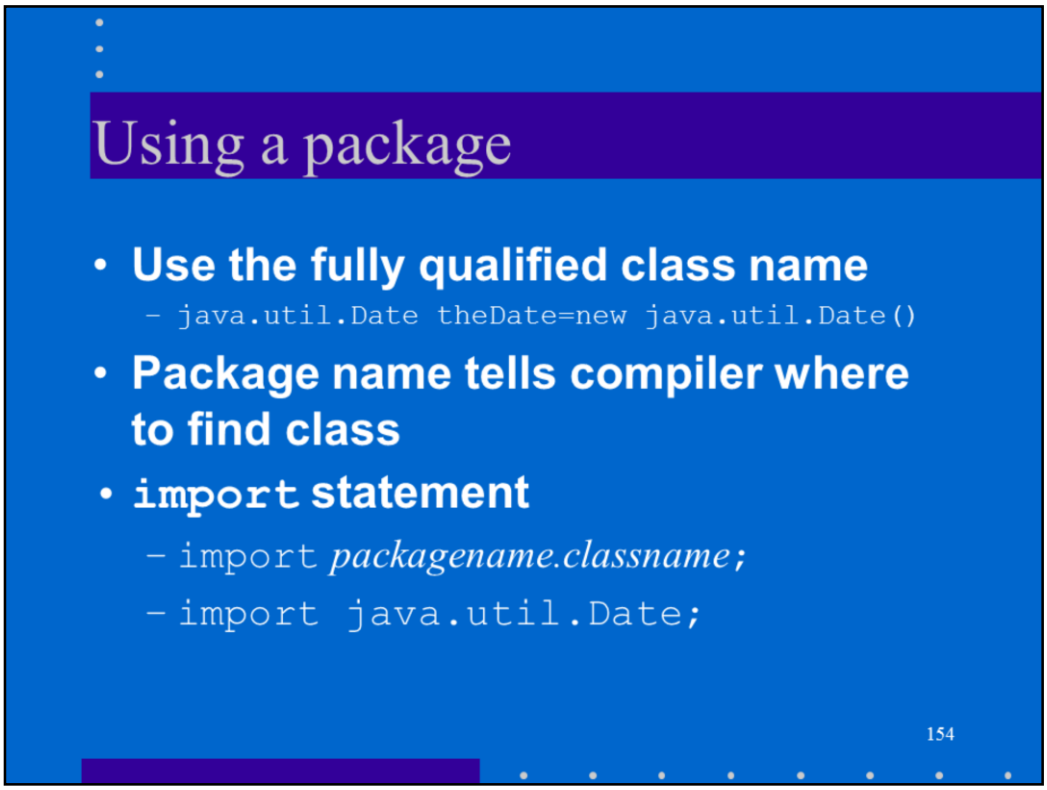
The Java Specification defines packages in the following way:

Java programs are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. A type is accessible outside the package that declares it only if the type is declared public.

The naming structure for packages is hierarchical . The members of a package are class and interface types , which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

A package can be stored in a file system or in a database . Packages that are stored in a file system have certain constraints on the organization of their compilation units to allow a simple implementation to find classes easily. In either case, the set of packages available to a Java program is determined by the host system, but must always include at least the three standard packages java.lang, java.util, and java.io. In most host environments, the standard packages java.applet, java.awt, and java.net, which are not described in this specification, are also available to Java programs.

A package consists of a number of compilation units . A compilation unit automatically has access to all types declared in its package and also automatically imports each of the types declared in the predefined package java.lang.


Using a package
• Use the fully qualified class name
– java.util.Date theDate=new java.util.Date()
• Package name tells compiler where to find class
• import statement
– import packagename.classname;
– import java.util.Date;
154

There are two ways to reference an item in a package.

Items may be given a fully qualified name. For example the class `java.util.Date()` references a constructor in a package called java.util.

The name indicates to the compiler that the definition of the class will be found in the `util` subdirectory of a directory called `java`. The full pathname of the directory where the class is found will depend on the classpath environment variable (see later).

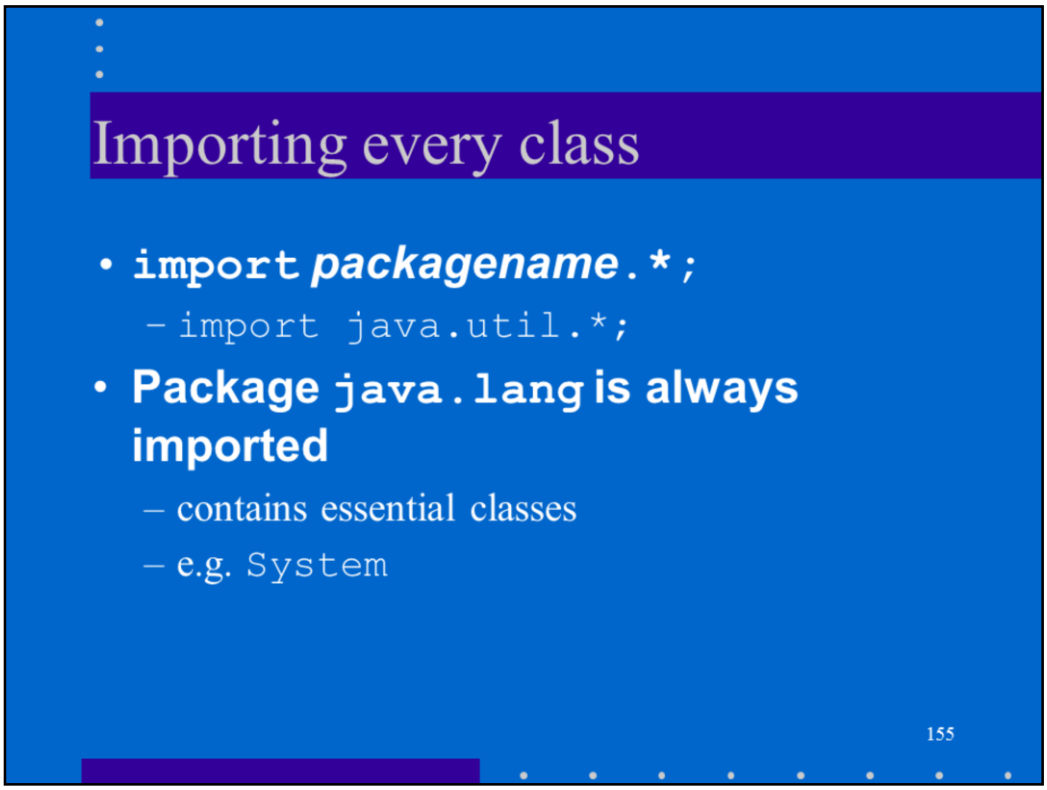
The other way a Java application or applet may reference classes defined outside of the file directory resides in is to use the `import` statement. The `import` statement informs the compiler that an external class definition is being used and, as in the fully qualified version, tells the compiler where to find the definition.

If the statement

```
import java.util.Date;
```

appears at the beginning of a file containing Java classes, then the class `Date` and all its public methods and public instance variables can be used without full qualification.

`import` statements must appear before any class definitions in any file where they are used.



The `util` subdirectory of the `java` directory may of course contain many class definitions. It is possible to make all of these available to a Java applet or application by using a * as the last element of the `import` statement.

```
import java.util.*;
```

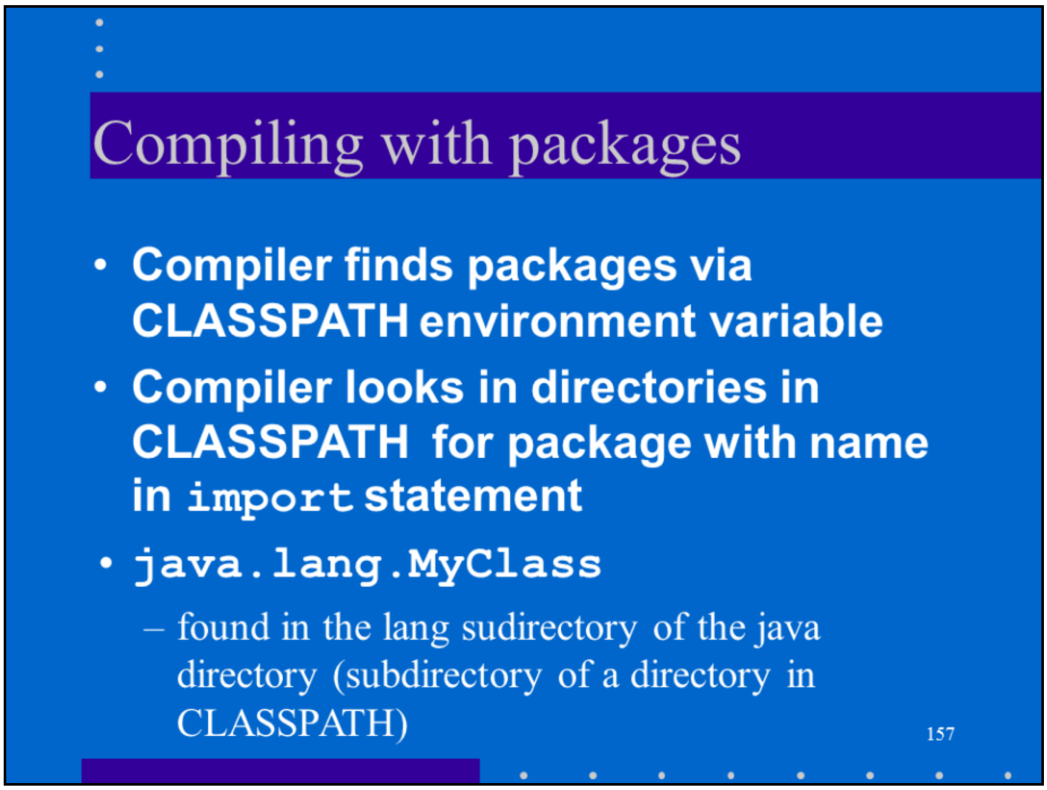
means that all the class definitions in the package `java.util` are available.

One package `java.lang` is imported by default into any Java application or applet (it is not necessary to explicitly import it). This package contains class definitions necessary for any Java application or applet e.g. the definition of class `Object`.

The standard packages in Java 1.1 are listed on the slide.

Java packages may be compressed by storing them as *jar* files. The Java compiler can access class definitions from these files without needing to decompress them.

Normally, after installing the JDK on a machine a directory called (in the case of Java 1.4.0) `jdk1.4.0` is created. The standard classes normally reside in the directory `jdk1.4.0/jre/lib/rt.jar`


Compiling with packages
• Compiler finds packages via CLASSPATH environment variable
• Compiler looks in directories in CLASSPATH for package with name in import statement
• java.lang.MyClass
– found in the lang sudirectory of the java directory (subdirectory of a directory in CLASSPATH)
157

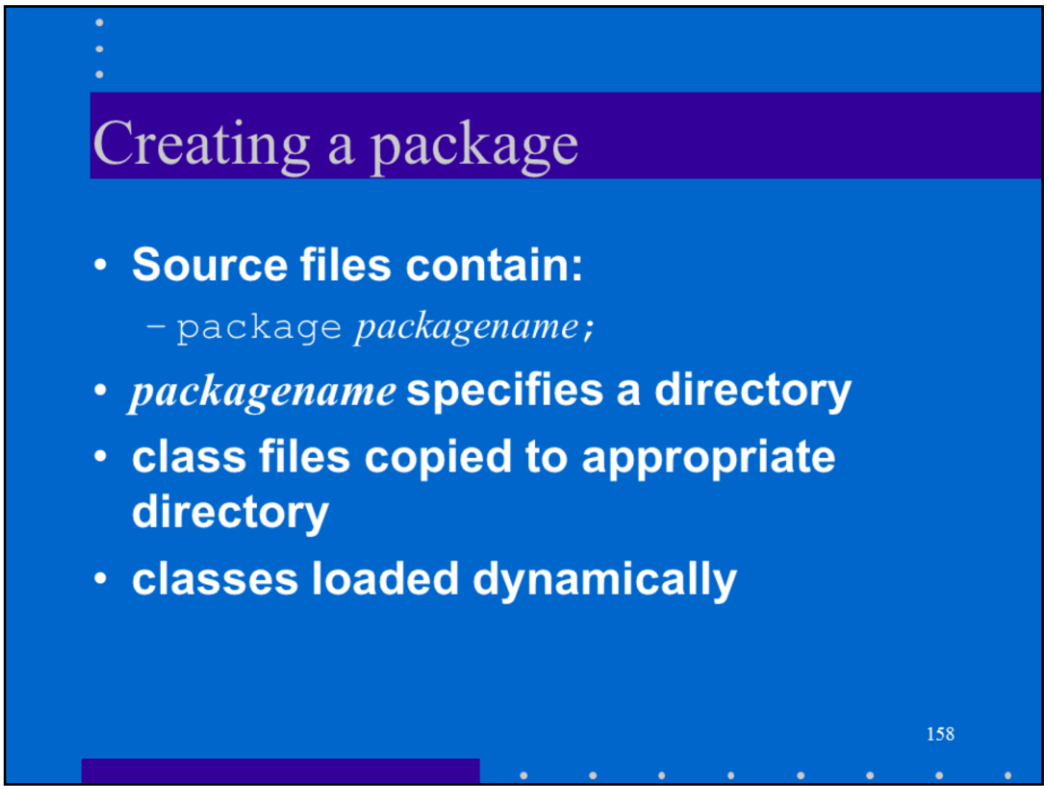
Sometimes, before running the Java compiler on any platform it is necessary to specify to set up a CLASSPATH environment variable. In Unix systems running the Bourne shell it is done in the `.profile` file.

The CLASSPATH environment tells the Java compile which directories to look in to find packages.

The –cp directive of the compiler allows a temporary classpath to be specified.

```
import java.lang.MyClass;
```

instructs the compiler to look for a `lang` subdirectory of a subdirectory of the directories on the CLASSPATH.



To define a package it is necessary to insert a package statement at the very beginning of the source file which contains the definitions of classes which will form part of the package. The package statement simply names the package the class is part of and at the same time defines the directory structure in which it will be stored.

package useful.mystuff

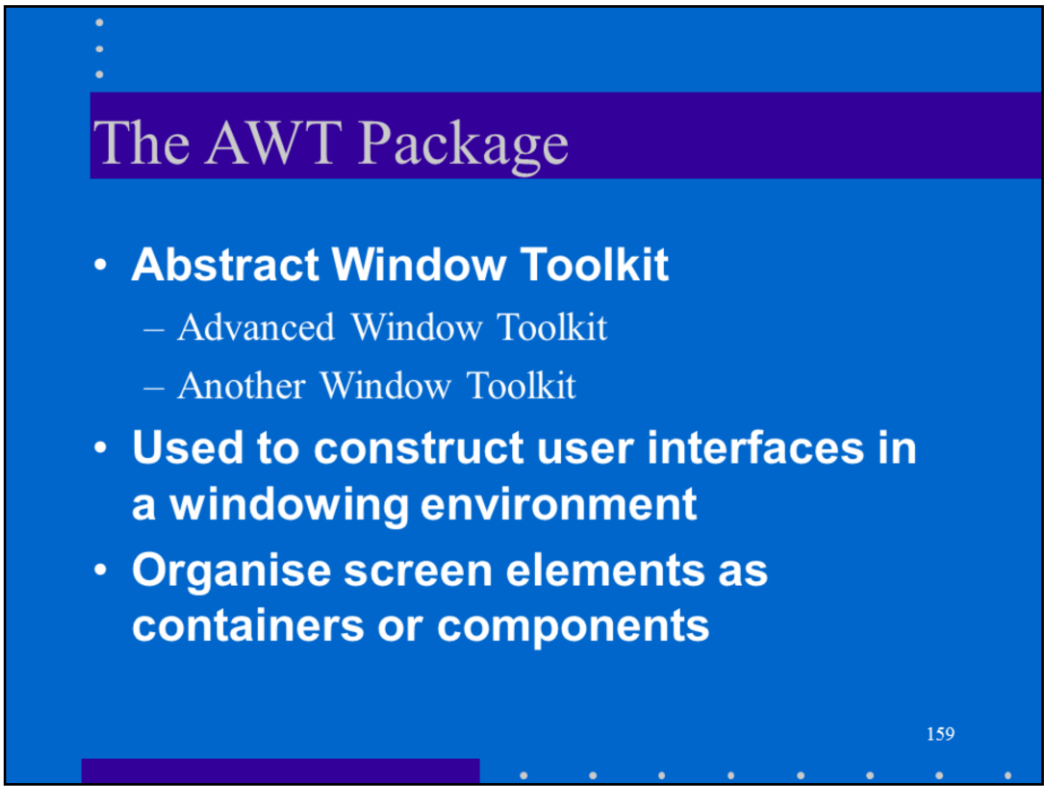
defines a package mystuff and states that it will be stored in a directory mystuff which is a subdirectory of useful.

The programmer is responsible for setting up the correct directory structure and copying the class files into it. It is possible to do this automatically at compile time with the -d option of javac.

javac -d /home/avh/classes MyProgram.java

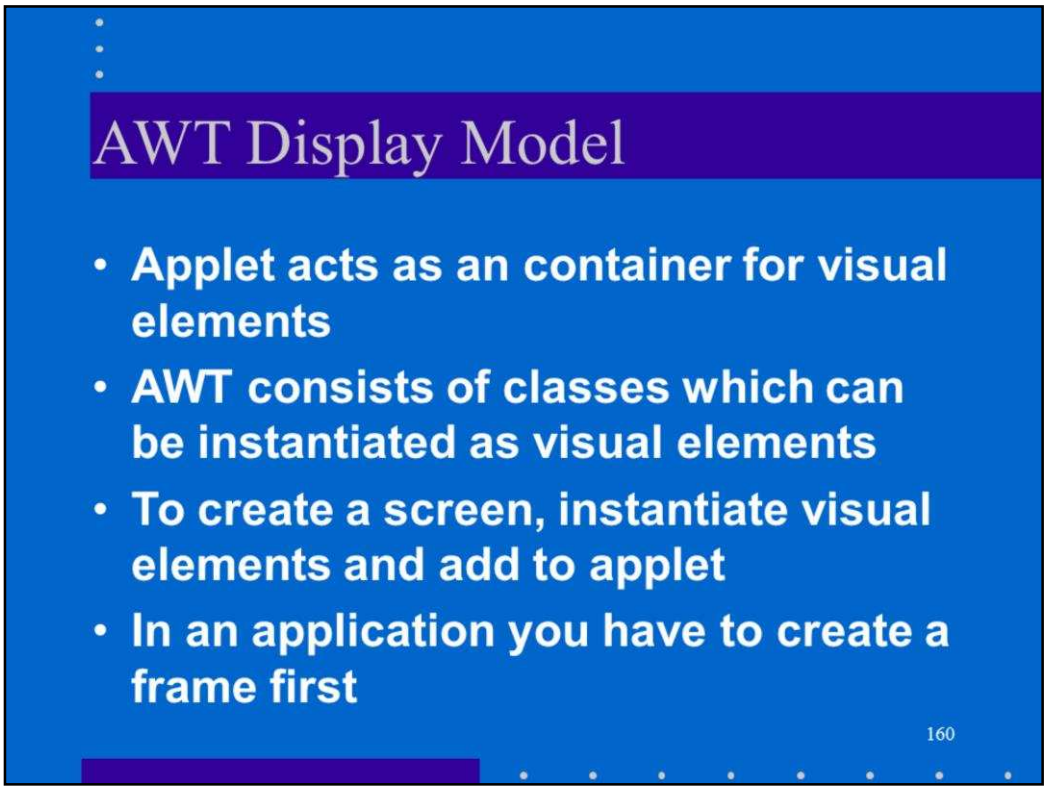
causes the compiled class files for the classes in the MyProgram.java source file to be saved in the directory /home/avh/classes. If your classes are defined in the package demos/awt, the class files would be placed in the directory /home/avh/classes/demos/awt.

In C++ code in libraries (except for dynamic load libraries) is physically added to the binary files during compilation. In Java this is not the case, compiled code is loaded as and when it is needed. Consequently, if a package is imported into an application or applet and the directory structure is subsequently altered then an error will occur. Similarly, if an application is to be ported to another machine then the packages it uses must also be ported.


The AWT Package
Abstract Window Toolkit
– Advanced Window Toolkit
– Another Window Toolkit
Used to construct user interfaces in a windowing environment
Organise screen elements as containers or components
159

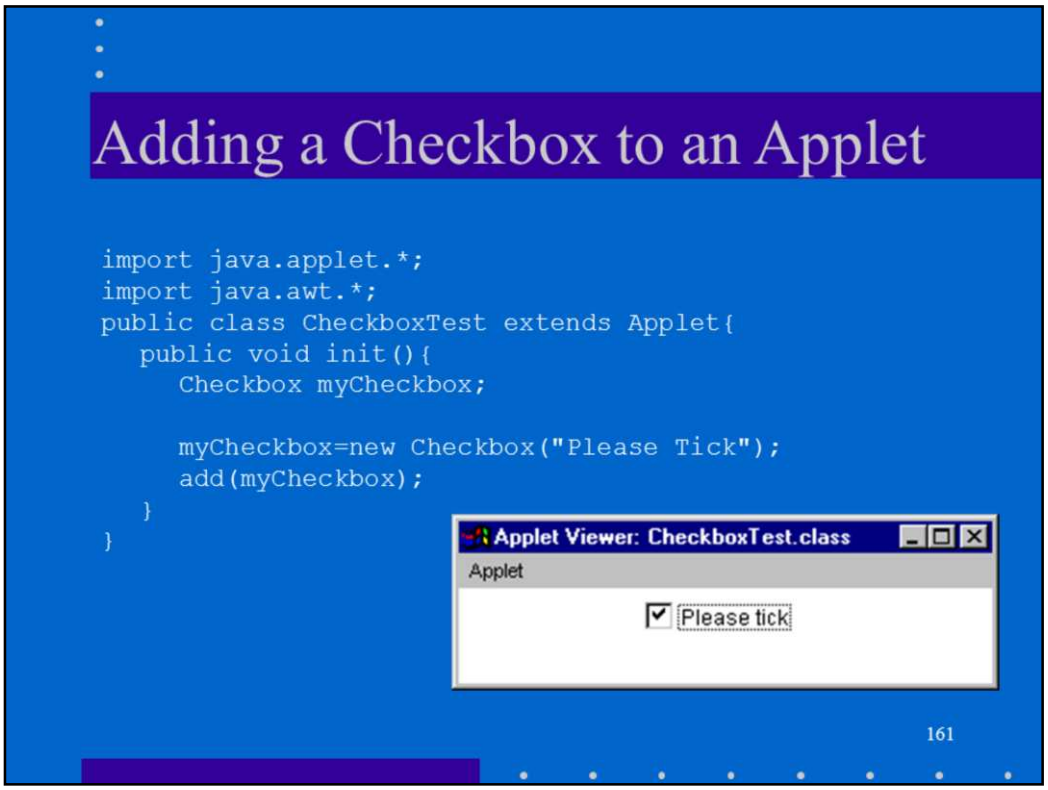
The package provided in order to make it possible to communicate with a windowing environment is called the AWT or Abstract Window Toolkit. It is called Abstract because it is independent of any particular operating system's windowing mechanism. The same code (source or binary) will work on Solaris, Windows, MacOs or whatever. In each case the graphic elements will take on the look and feel of the platform they are executing on.

AWT is a set of components and containers. Containers are essentially objects which hold components. The most obvious container is a window. In a window we can place components such as buttons, slider bars and check boxes.


AWT Display Model
• Applet acts as an container for visual elements
• AWT consists of classes which can be instantiated as visual elements
• To create a screen, instantiate visual elements and add to applet
• In an application you have to create a frame first
160

An `Applet` is a container in which visual components can be placed. Class Applet inherits from the `Container` class. The AWT contains many classes which can be instantiated as objects which may represent visual elements. Once these objects have been instantiated they are written to the screen by adding them to the container class.

An application is not a container. Therefore to use AWT within an application the application must create a container object. This is invariably a `Frame` which in Java is the equivalent of a window in Microsoft Windows.



This example shows how an applet may have a visual element added to it.

The applet imports the packages `java.applet` (to provide the definition of class Applet) and `java.awt` (to define the visual element classes).

`myCheckbox` is defined as having type `Checkbox`.

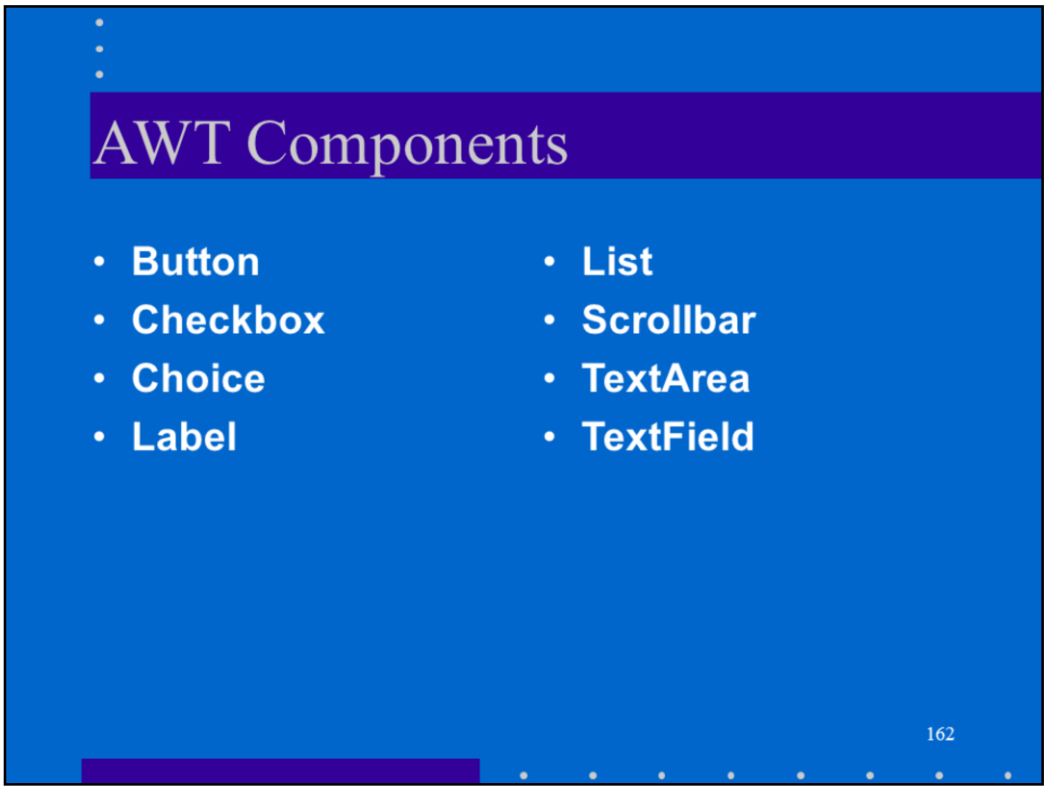
A `Checkbox` object is instantiated with the new operator using a constructor which labels it with the string "Please Tick".

Applet is a subclass of `Container` and inherits the `add` method from it. This method expects an object of type `Component`. `Checkbox` inherits from `Component` and so is of the correct type.

The applet will compile to `CheckboxTest.class` and can be loaded by the following HTML (in my case stored in CheckboxTest.html)

```
<html>
<head><title>Checkbox</title></head>
<body>
<applet code=CheckboxTest.class width=300
height=50>
</applet>
</body>
</html>
```

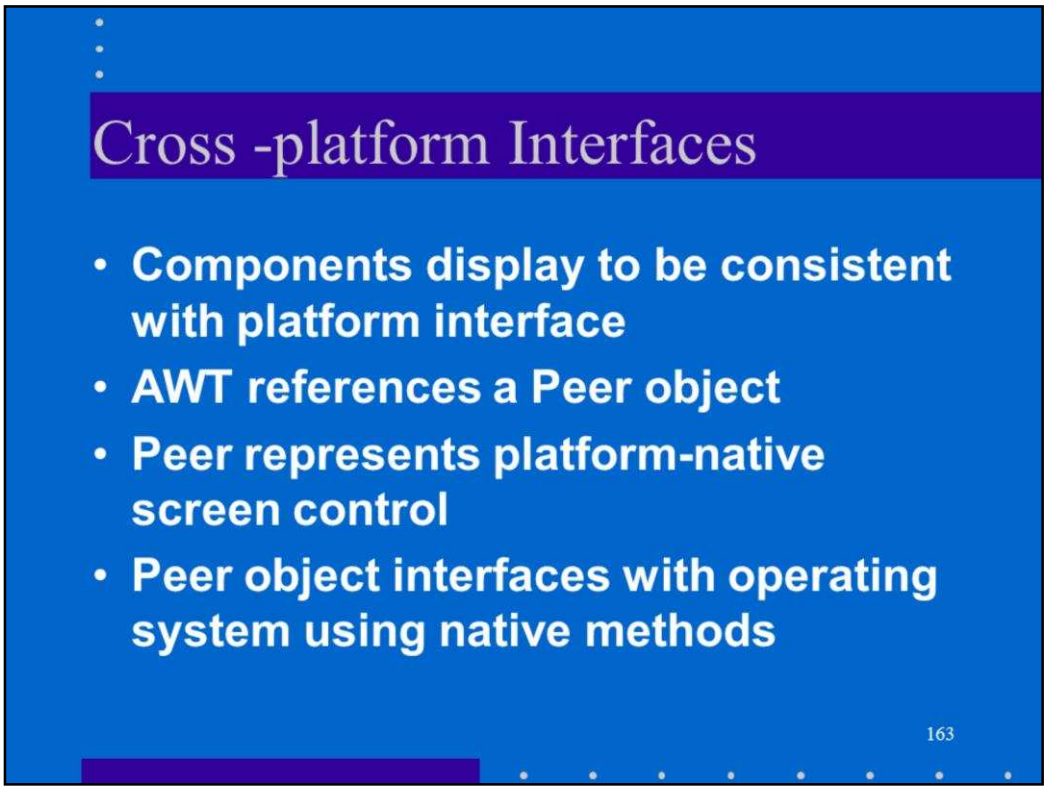
This is run by executing `appletviewer CheckboxTest.html`



The components available in the AWT are very much what one would expect to be available in a basic windowing system. The ones listed above will described in more detail later in the lecture.

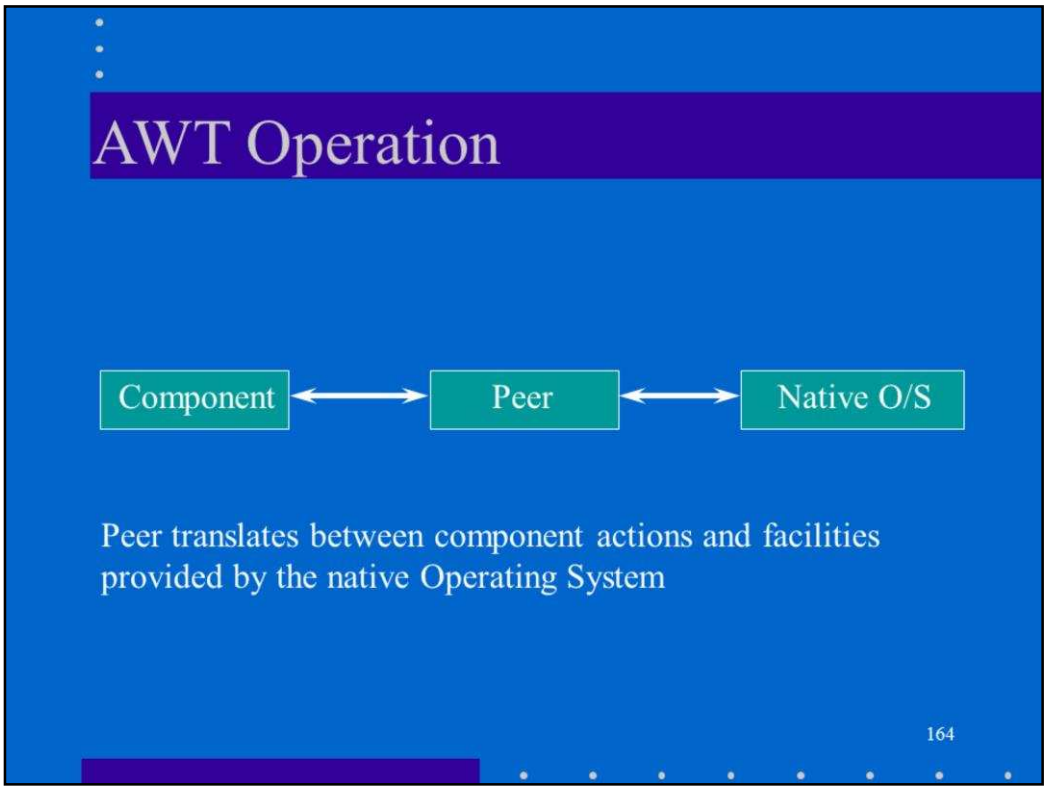
Full details of all the AWT components and containers are available at

http://java.sun.com/j2se/1.4/docs/api/index.html


Cross -platform Interfaces
• Components display to be consistent with platform interface
• AWT references a Peer object
• Peer represents platform-native screen control
• Peer object interfaces with operating system using native methods
163

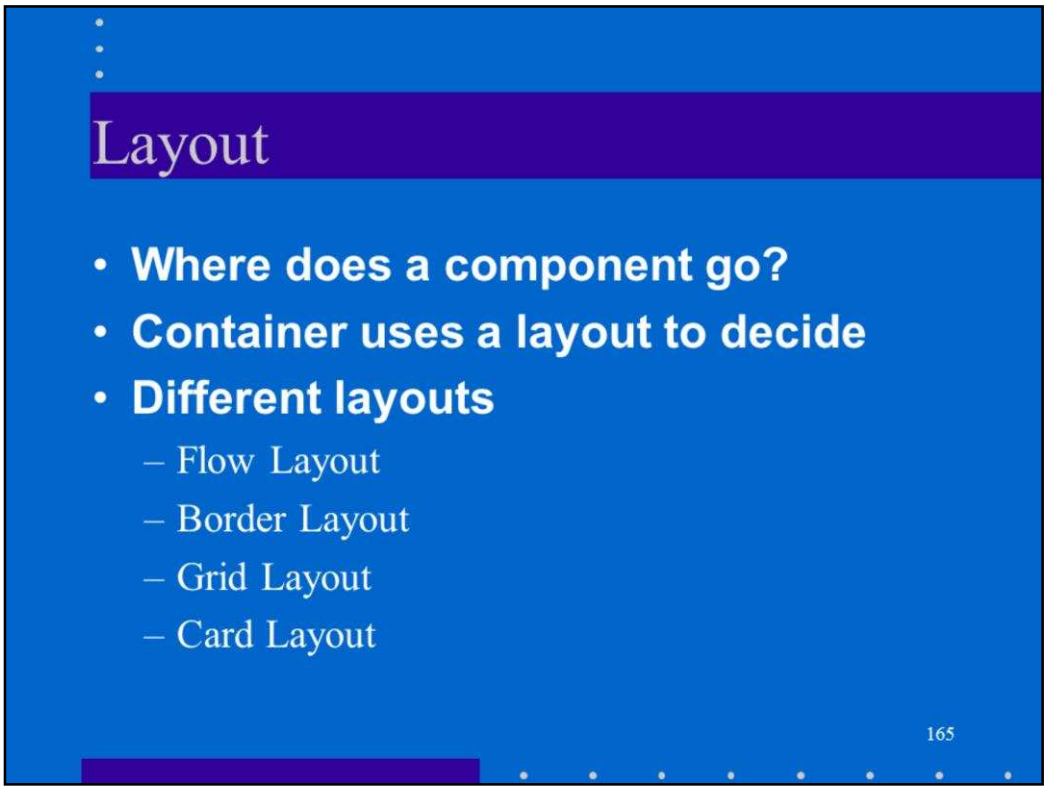
When a Java program runs it is the operating system it runs on which is responsible for drawing the various visual elements. This has an implication for the speed at which the program runs. As Java is an interpreted rather than a compiled language it runs more slowly than many other languages. C++ is said to be ten times faster in execution than Java. If a Java program is required to undertake lots of computation then this is a fair comparison. If, however, the program is mostly concerned with driving a user interface then the speed differential is less important. In this case both the Java application and the C++ program will spend the majority of their time calling operating system routines which execute at the same speed regardless of whether they are called from Java or C++. The speed differential is therefore much less important.

Communication with the operating system takes place in the following way. Each AWT object stores a reference to a Peer object. The Peer object represents a platform native on-screen visual component. It manipulates this component using platform dependent code. Whenever the AWT receives a message it passes it on to the Peer object for execution.



Essentially the Peer object code is there to make the graphical part of the host operating system look like an object!

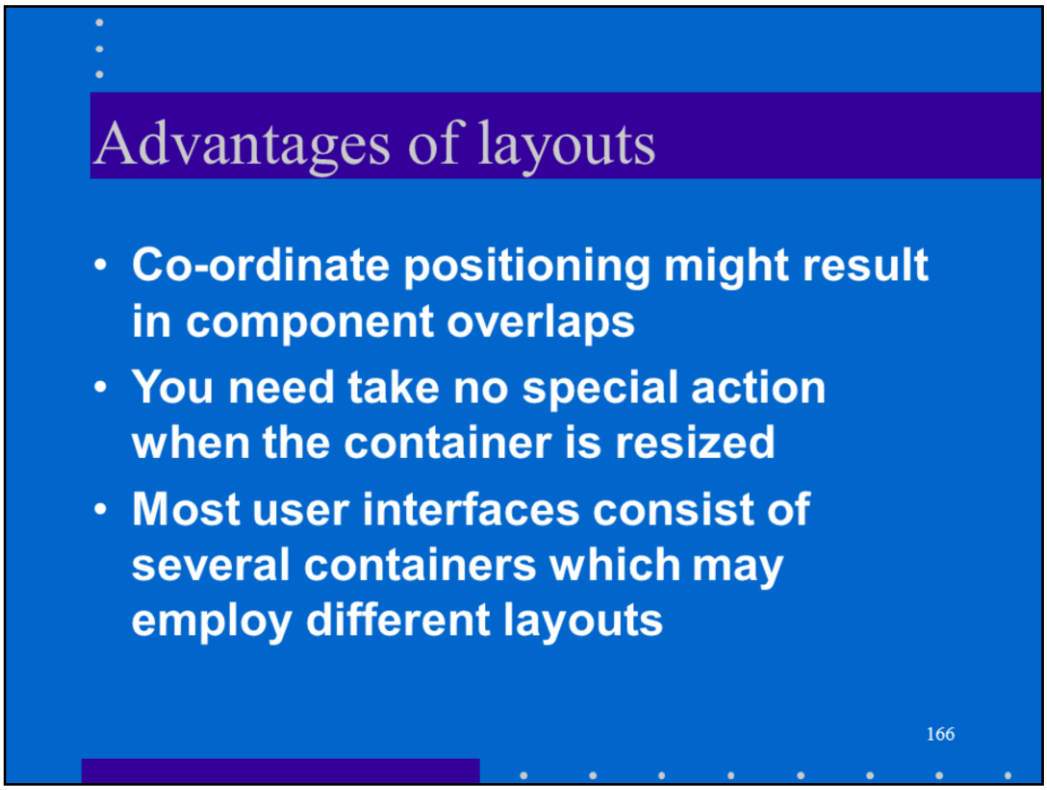
There are a couple of disadvantages with the AWT. As it insists that everything must look the same to the programmer there are occasions where on a particular windowing system an inefficient approach has to be taken. Also AWT works on the basis of lowest common denominator. It only provides those features which are common to all windowing systems. MS Windows has lots of useful visual features which are not supported by AWT simply because none of the other windowing environment support them. This second point is one which seriously threatens Java's penetration of the marketplace.



So far we have identified that a window can be represented by a container and that a component can be added to it. We have seen an example of this. But which part of a window will a component go in to when it is added to it?

Java allows precise positioning of elements within windows. This approach, however takes considerable effort on the part of the programmer. A much easier way is to associate a layout with a container.

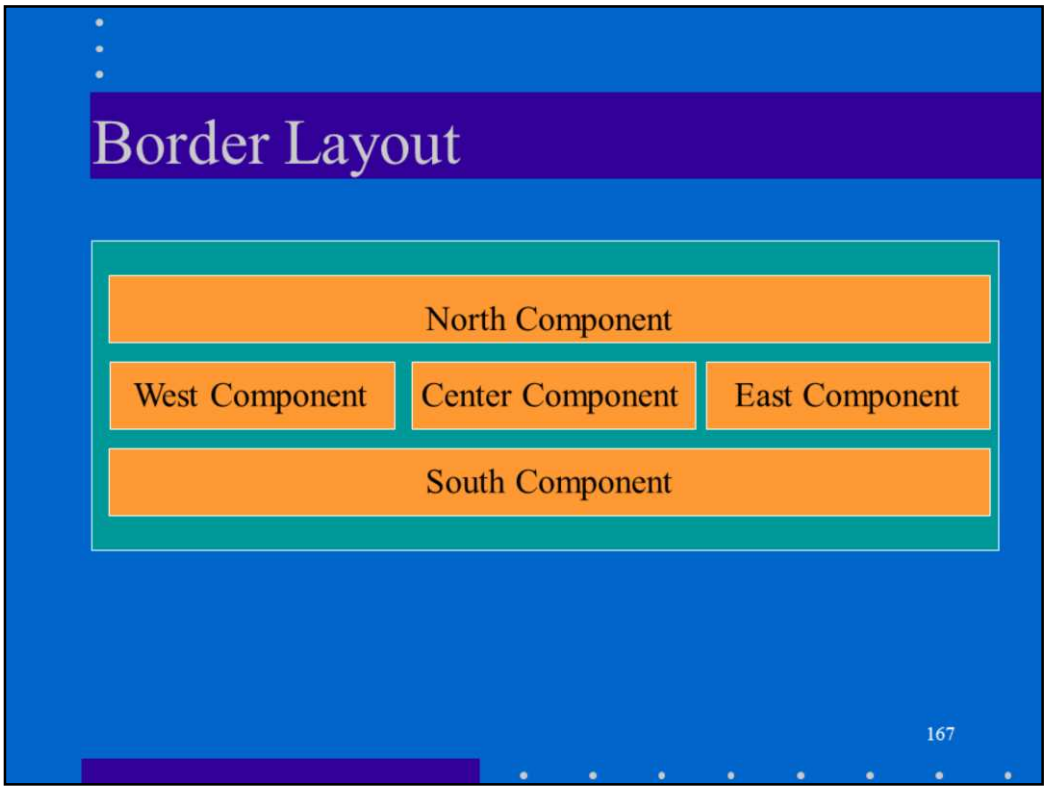
Layouts simply specify an approximate position for components. Different layouts require different addition methods. Java provides four simple layouts: flow layout, border layout, grid layout and card layout. More complex layouts such as grid bag layout are also available and it is possible, via inheritance, to design your own layouts.


Advantages of layouts
• Co-ordinate positioning might result in component overlaps
• You need take no special action when the container is resized
• Most user interfaces consist of several containers which may employ different layouts
166

If programmers associate components with window co-ordinates then, unless they are careful to consider the size of those components, it will be possible for the components to overlap.

When a window is resized the components must react to the size change. For example consider a small window with six buttons which just fill it. If the window is enlarged we expect a larger window with six buttons in the corner or the same window with six larger buttons more widely spaced. If the second of these is required then the co-ordinates must be recalculated. Layouts take care of this problem.

Although the list of layouts seems limited, it is possible to embed containers (Panels) within layouts and create windows which contain a variety of layouts. There is plenty of opportunity for variation.



This is the border layout which allows the programmer to place five components. Each component is placed in the positions labelled North, South, West, East or Center (spelt in the American way!).

Example of Border Layout

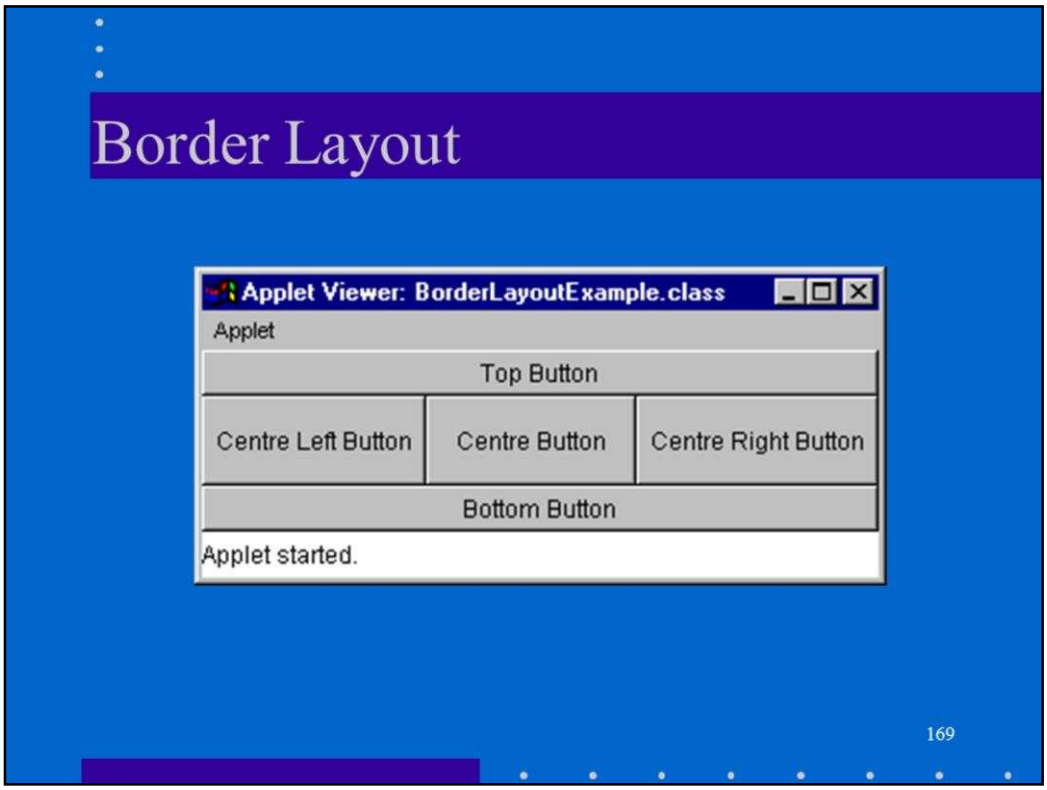
```
import java.applet.*;
import java.awt.*;
public class BorderLayoutExample extends Applet{
   public void init(){
        setLayout(new BorderLayout());
        add("South",new Button("Bottom Button"));
        add("North",new Button("Top Button"));
        add("West",new Button("Centre Left Button"));
        add("East",new Button("Centre Right Button"));
        add("Center", new Button("Centre Button"));
   }
}
```

168

This code shows an applet which lays out five button components using the border layout. The applet imports the `java.applet` and the `java.awt` packages. As with all applets it inherits from class `Applet`.

The only method declared overrides the `init()` method of `Applet`. It calls the `setLayout()` method of `Container` with a parameter which is a `BorderLayout` object.

Five buttons are created with labels and are added to the appropriate place in the layout. In this case the `add()` method has a `String` parameter which is used to specify into which of the five positions the buttons will go.



The following HTML is necessary to run the applet:

```
<html>
<head><title>Checkbox</title></head>
<body>
<applet code=BorderLayoutExample.class width=300 height=75>
</applet>
</body>
</html>
```

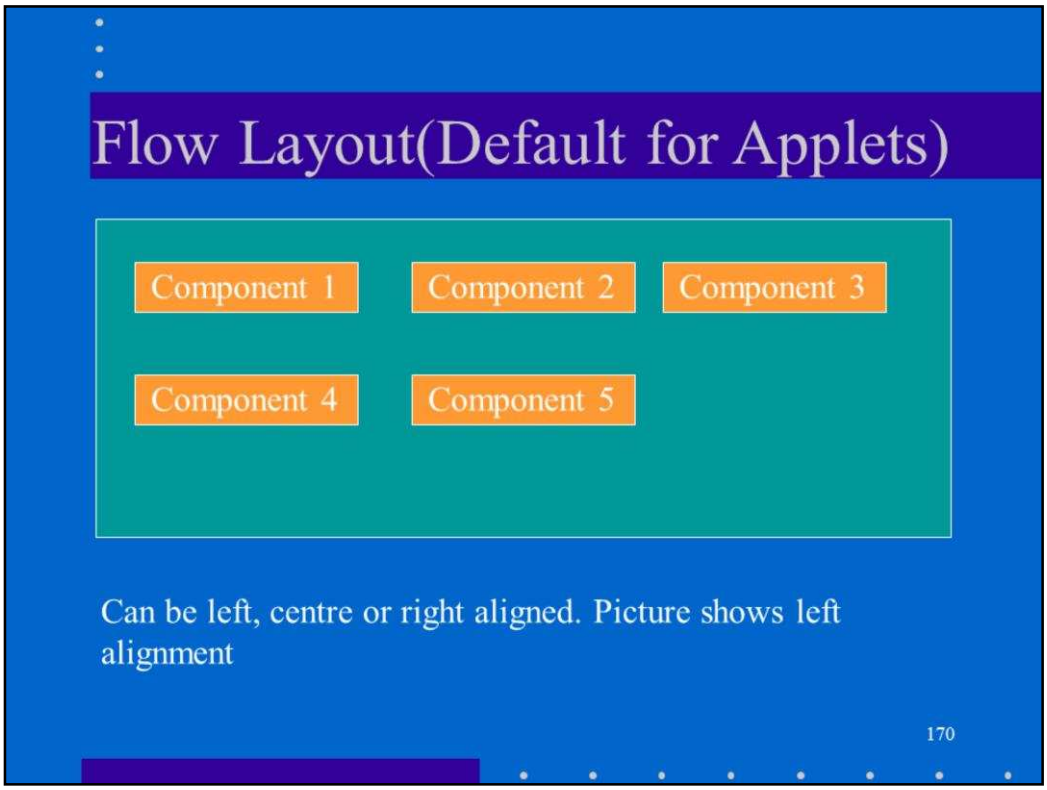
It is executed by typing:

```
appletviewer BorderLayoutExample.html
```

on the command line.

On loading the `init()` method is invoked and the window is drawn as shown.

If the window is resized the buttons will expand in size.



If a layout is not specified, components are placed using flow layout.

Flow layout simply places a component in the first available space, the second in the next available space and so on. Placement starts at the top left hand corner and proceeds to the right until that line is full after which the next line is chosen.

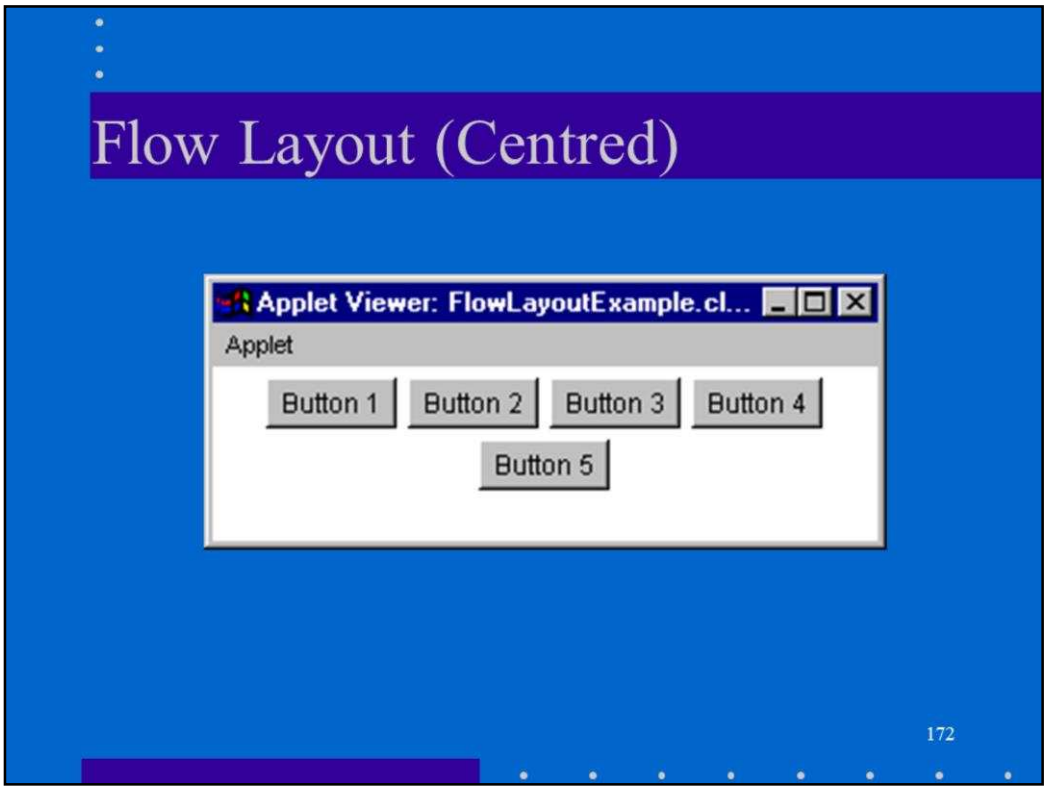
Flow control can be adapted so that a line of components is left, right or centre aligned. The default is centre aligned. You cannot choose where something is placed, the order in which it is added to the container determines this.

Example of Flow Layout

```
import java.applet.*;
import java.awt.*;
public class FlowLayoutExample extends Applet{
    public void init(){
        setLayout(new FlowLayout());
        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));
    }
}
```

171

This applet is similar to border layout example except that here `FlowLayout` is set. In this layout it is no longer possible to specify a position when invoking `add()`.



The HTML to run this is as follows:

```
<html>
<head><title>Checkbox</title></head>
<body>
<applet code=FlowLayoutExample.class width=300 height=75>
</applet>
</body>
</html>
```

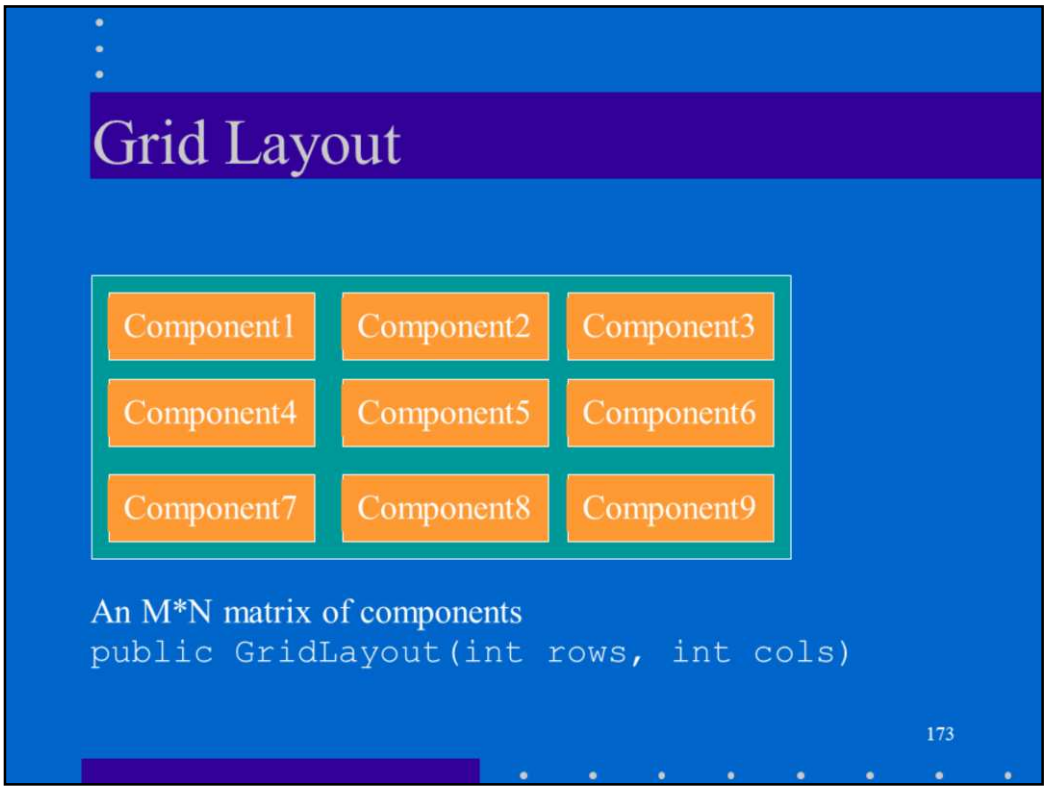
It is executed by typing:

```
appletviewer FlowLayoutExample.html
```

on the command line.

In the example shown, four buttons have been added and have filled the first line. Button five has been placed on the second line and, because the default alignment is centred, it appears in the centre of the line.

With this layout, if the window is widened the buttons remain the same size and will eventually fit on the top line.



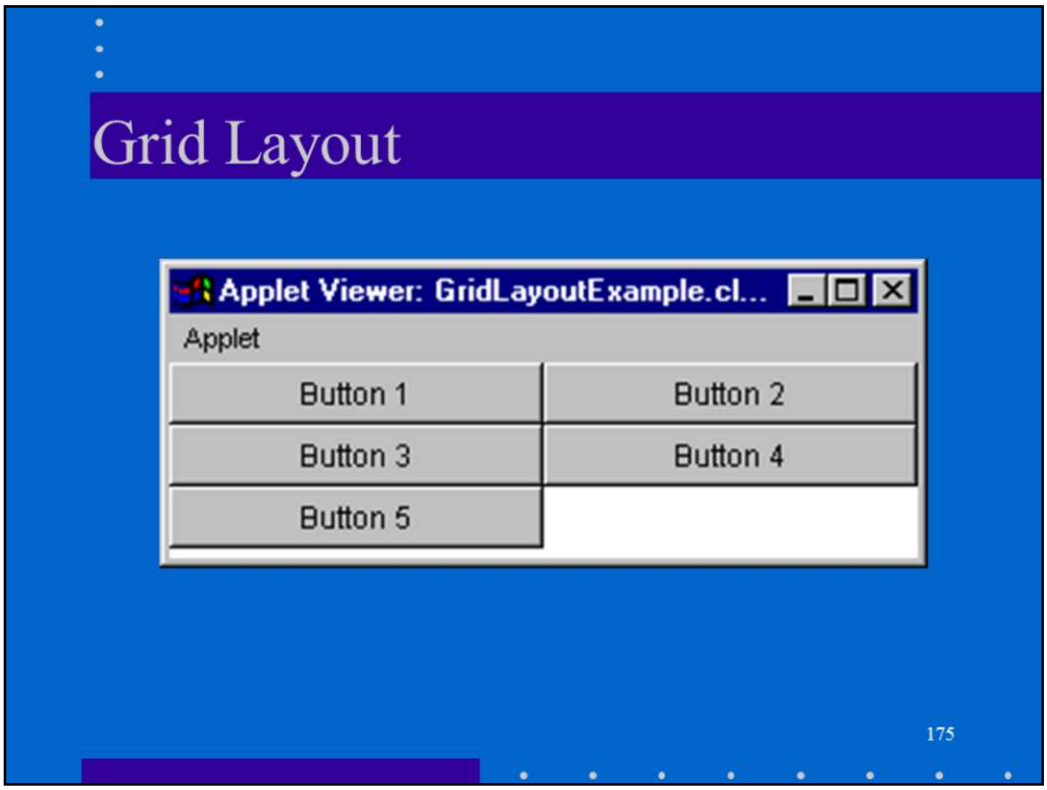
A grid layout is similar to a two dimensional array of visual elements. The layout has to be given a size in terms of number of rows and number of columns when it is created. Unlike a two dimensional array, however, you cannot place visual components where you choose in the grid. Components fill up the first row, left to right, when that is full the second is begun. This is very much the same as flow layout with left alignment, however, the difference is that when a window with a grid layout is resized the components stay in the same relative positions and grow in size whereas in flow layout they stay the same size and change position.

Grid Layout Example

```
import java.applet.*;
import java.awt.*;
public class GridLayoutExample extends Applet{
   public void init(){
        setLayout(new GridLayout(3,2));
        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));
   }
}
```

174

In this example the grid layout is set to have three rows and two columns. Five buttons are added in much the same way they would be added to a flow layout. In a flow layout however the look of the resulting window in terms of button positioning would depend on the size of the window. With a grid layout we are guaranteed that we will see two columns of buttons. The size of the button is determined by the window size.



The HTML necessary to run this applet is:

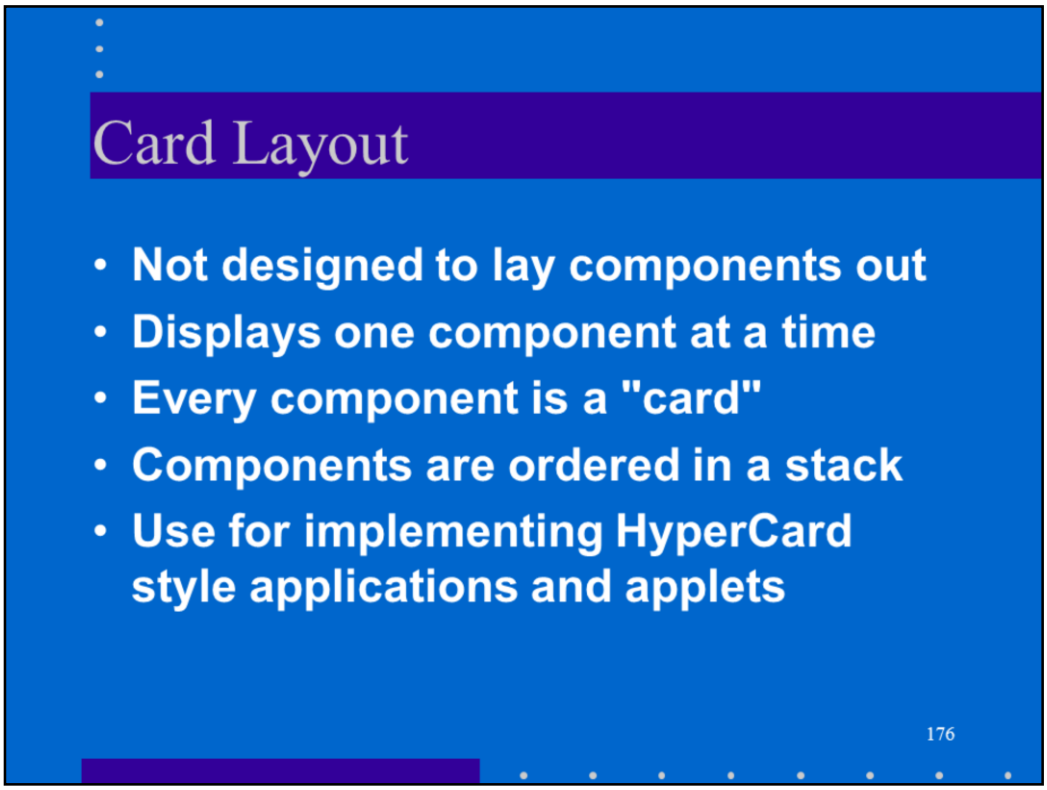
```
<html>
<head><title>Checkbox</title></head>
<body>
<applet code=GridLayoutExample.class width=300 height=75>
</applet>
</body>
</html>
```

The applet can be run by typing

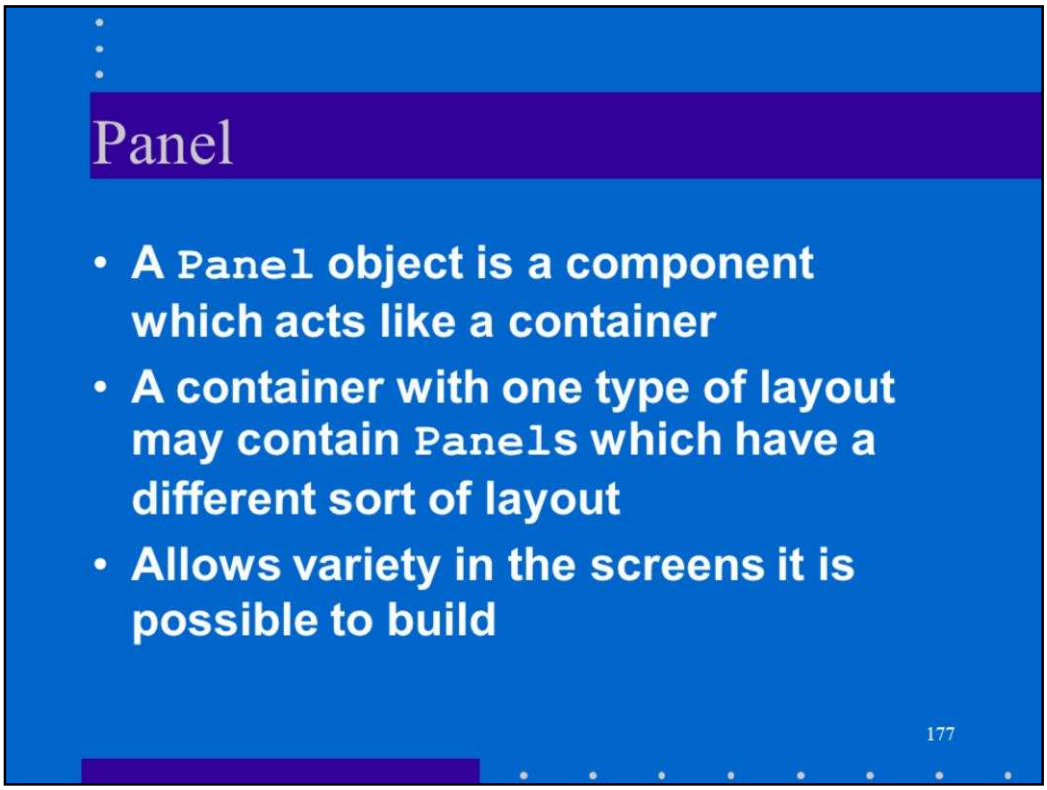
```
appletviewer GridLayoutExample.html
```

at the command line

Only five buttons have been added so the final grid position remains empty.


Card Layout
• Not designed to lay components out
• Displays one component at a time
• Every component is a "card"
• Components are ordered in a stack
• Use for implementing HyperCard style applications and applets
176

This a layout which can be made to simulate some of the features of the tabbed dialogues in Windows '95. What it gives the programmer is a stack of canvases which can be accessed by name or by sequentially moving through the stack of cards. Card Layout on its own is not much use, however when panels (see next slide) are added to the cards then quite powerful dialogues can be created.


Panel
• A Panel object is a component which acts like a container
• A container with one type of layout may contain Panels which have a different sort of layout
• Allows variety in the screens it is possible to build
177

So far we have seen graphic elements which are either containers or components. A `Panel` is a component which may contain other components. A panel may therefore be used to enhance the on-screen appearance of a Java program. Each panel on a screen may have the same or different layouts selected for it. In effect, a panel is a subsection of a layout.

Panel Example

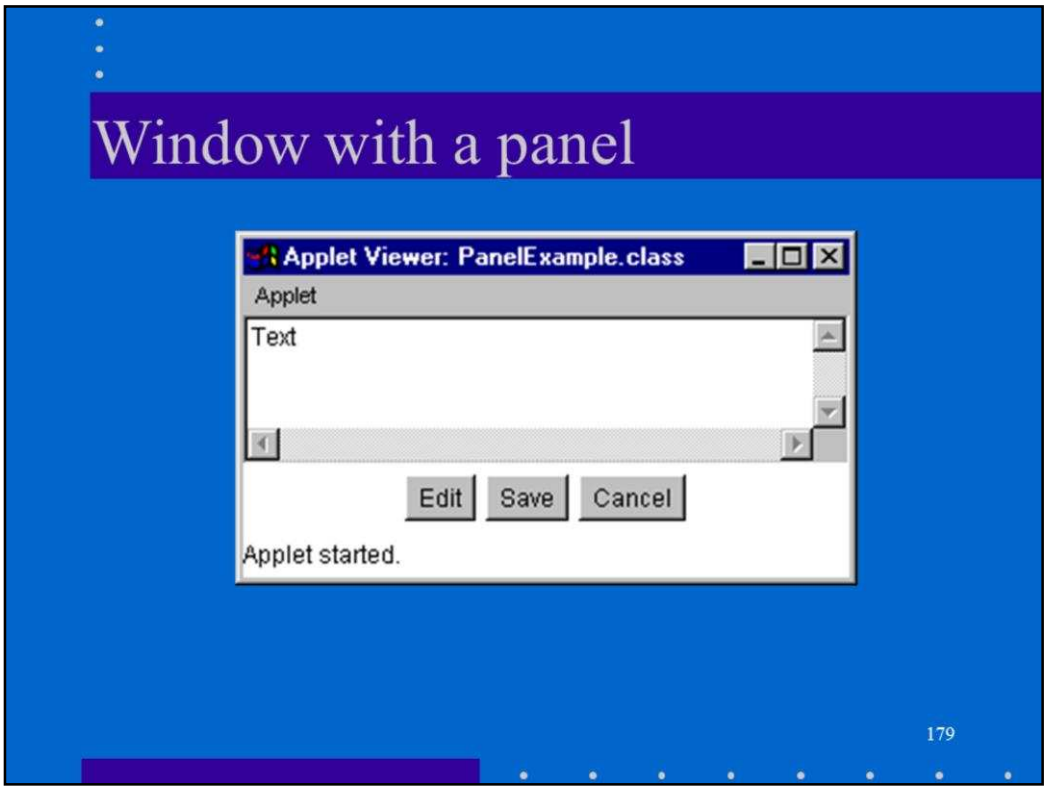
```
import java.applet.*;
import java.awt.*;
public class PanelExample extends Applet{
   public void init(){
        setLayout(new BorderLayout());
        Panel buttonBar = new Panel();
        buttonBar.setLayout(new FlowLayout());
        buttonBar.add(new Button("Edit"));
        buttonBar.add(new Button("Save"));
        buttonBar.add(new Button("Cancel"));
        TextArea myText = new TextArea("Text",5,40);
        add("South",buttonBar);
        add("Center",myText);
   }
}
```

178

In this example the applet uses a Border Layout. On its own a Border Layout allows the positioning of five graphic elements (north, east, south, west and centre). In this example the bottom element is set to be a `Panel`. As a panel is a container as well as a component it too may contain graphics elements. In the example shown three buttons are added to the panel which is referenced by the variable `buttonBar`.

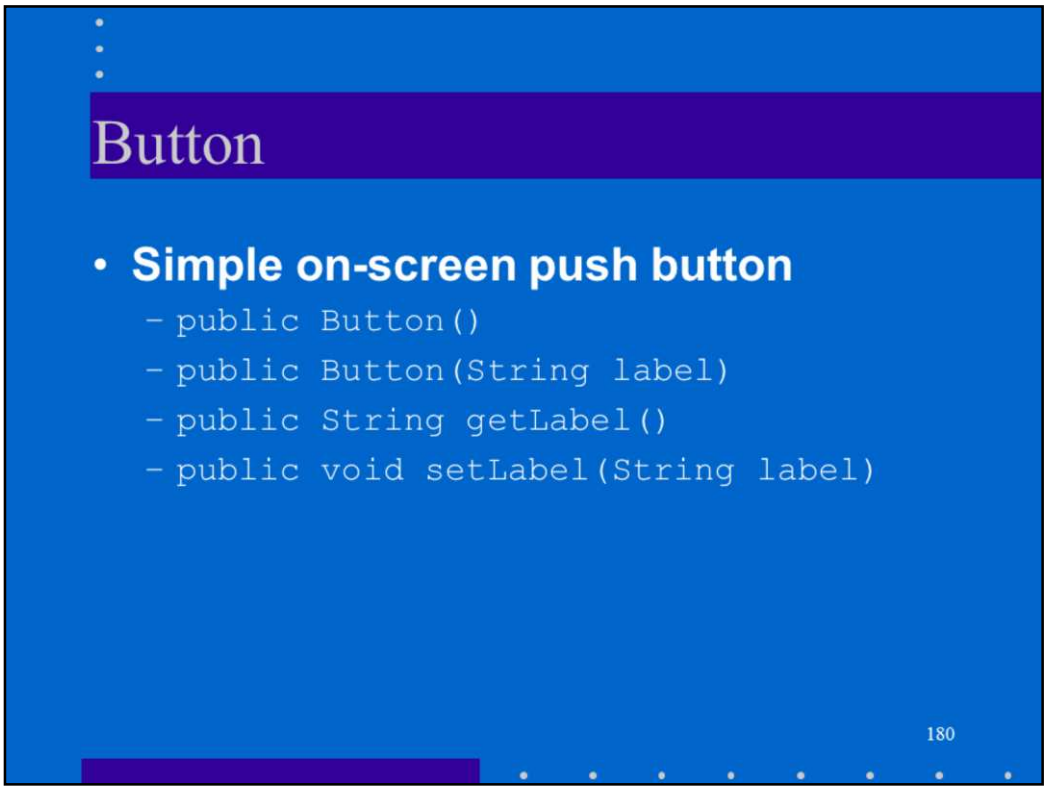
Note that adding to the panel requires full qualification as in `buttonBar.add` whilst adding to the main applet panel only requires the use of an unqualified `add` method.

The button bar panel uses a Flow Layout to organise its buttons.



The results of the code are shown here. As there aren't any north, west or east elements, the Text Area expands to fill the top part of the window. The panel in the south position contains the three buttons.

A Border Layout on its own would not be sufficient to achieve this, as only one button could appear in the south position. The use of a panel in this position allows a greater variety of effects to be created.




```
public class Button
extends  Component
```

This class creates a labelled button. The application can cause some action to happen when the button is pushed.

The gesture of clicking on a button with the mouse is associated with one instance of `ActionEvent`, which is sent out when the mouse is both pressed and released over the button. If an application is interested in knowing when the button has been pressed but not released, as a separate gesture, it can specialize `processMouseEvent`, or it can register itself as a listener for mouse events by calling `addMouseListener`. Both of these methods are defined by `Component`, the abstract superclass of all components.

When a button is pressed and released, AWT sends an instance of `ActionEvent` to the button, by calling processEvent on the button. The button's `processEvent` method receives all events for the button; it passes an action event along by calling its own `processActionEvent` method. The latter method passes the action event on to any action listeners that have registered an interest in action events generated by this button.

Event handling is described in Lecture 6.

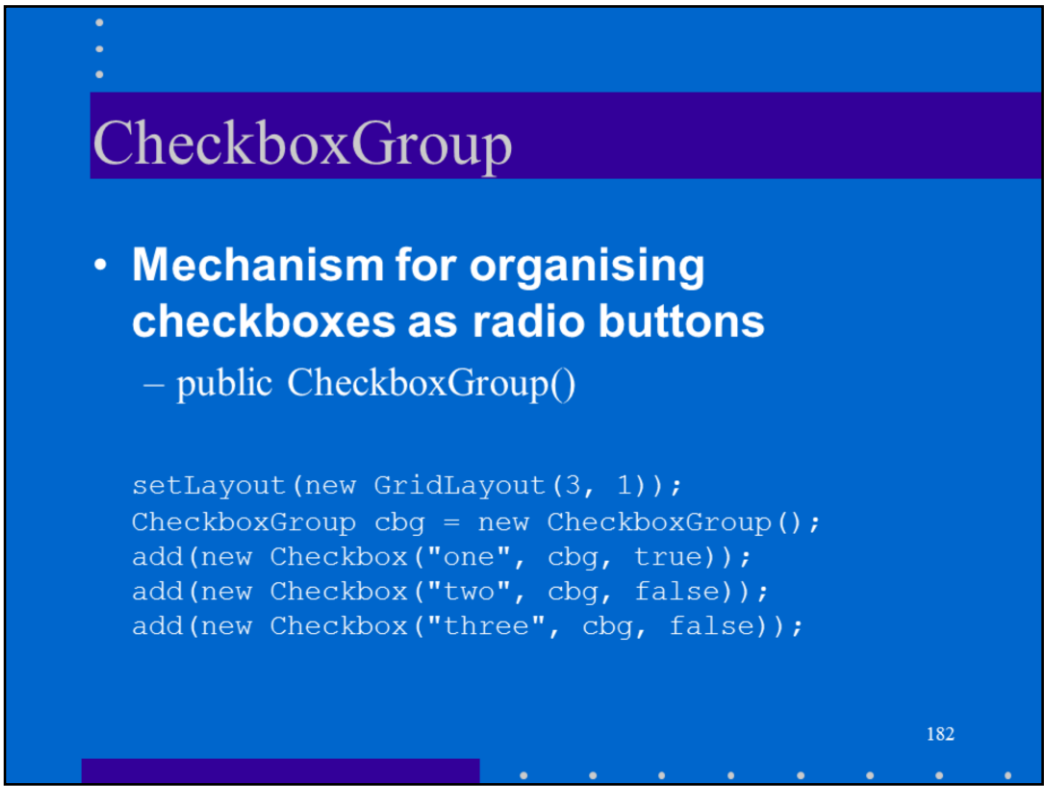
```
public class Checkbox
extends  Component
implements ItemSelectable
```

A check box is a graphical component that can be in either an "on" (true) or "off" (false) state. Clicking on a check box changes its state from "on" to "off," or from "off" to "on."

The following code example creates a set of check boxes in a grid layout:

```
setLayout(new GridLayout(3, 1));
add(new Checkbox("one", null, true));
add(new Checkbox("two"));
add(new Checkbox("three"));
```

Several check boxes can be grouped together under the control of a single object, using the `CheckboxGroup` class. In a check box group, at most one button can be in the "on" state at any given time. Clicking on a check box to turn it on forces any other check box in the same group that is "on" into the "off" state.



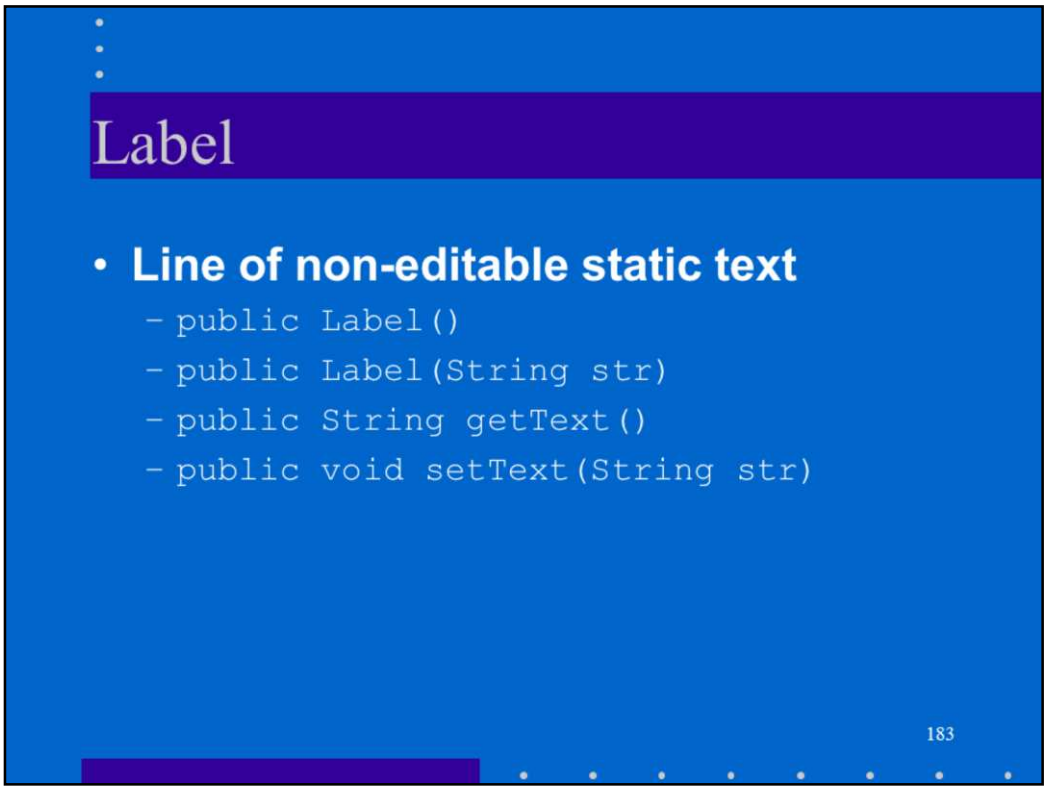

```
public class CheckboxGroup
extends  Object
implements  Serializable
```

The `CheckboxGroup` class is used to group together a set of `Checkbox` buttons.

Exactly one check box button in a `CheckboxGroup` can be in the "on" state at any given time. Pushing any button sets its state to "on" and forces any other button that is in the "on" state into the "off" state.

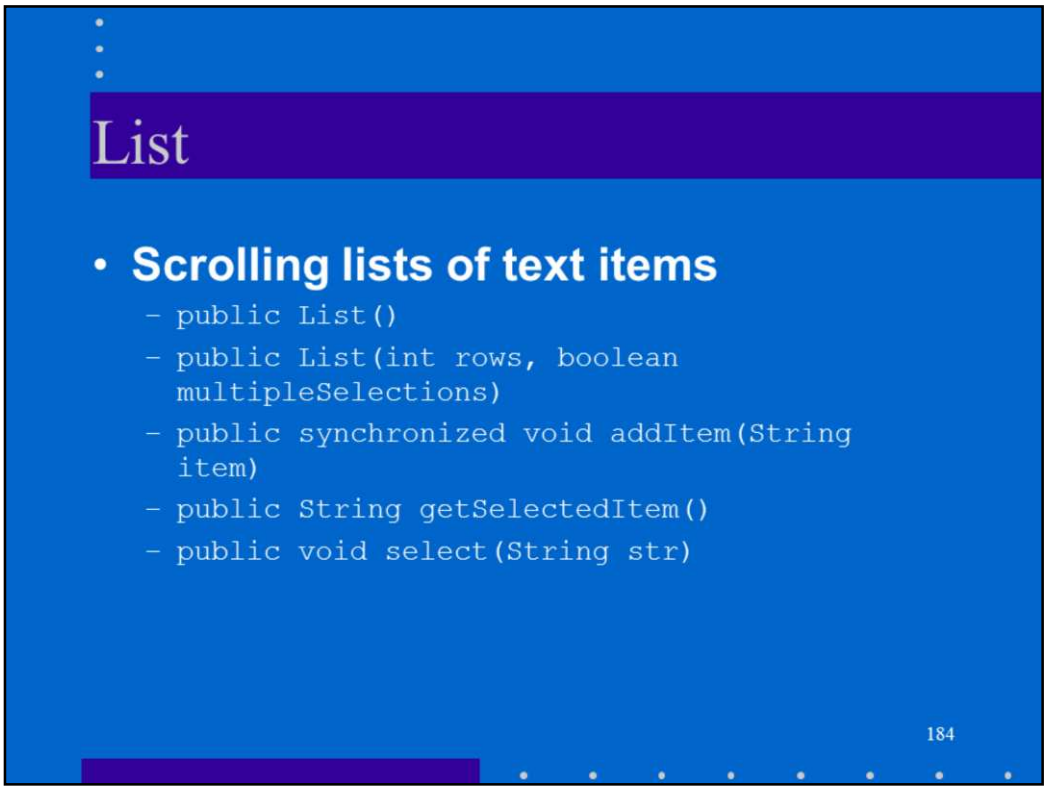
The following code example produces a new check box group, with three check boxes:

```
setLayout(new GridLayout(3, 1));
CheckboxGroup cbg = new CheckboxGroup();
add(new Checkbox("one", cbg, true));
add(new Checkbox("two", cbg, false));
add(new Checkbox("three", cbg, false));
```




```
public class Label
extends  Component
```

A `Label` object is a component for placing text in a container. A label displays a single line of read-only text. The text can be changed by the application, but a user cannot edit it directly.



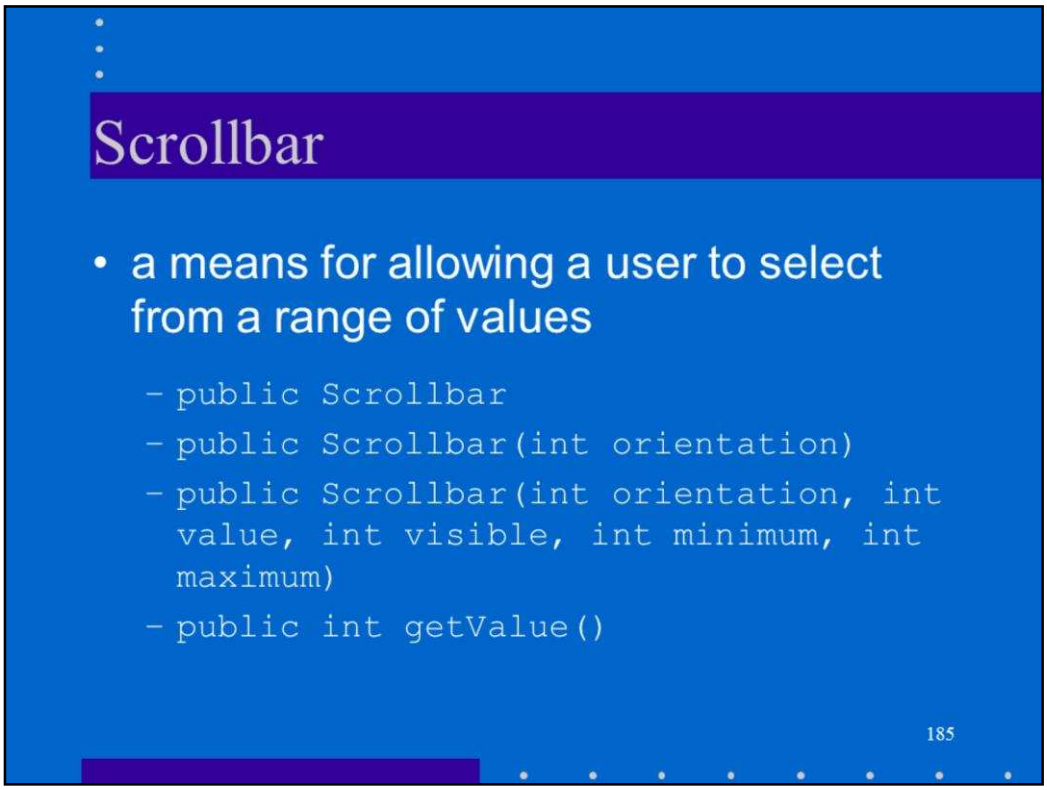

```
public class List
extends  Component
implements ItemSelectable
```

The `List` component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple items.

For example, the code . . .

```
List lst = new List(4, false);
lst.add("Mercury");
lst.add("Venus");
lst.add("Earth");
lst.add("Mars");
lst.add("Jupiter");
lst.add("Saturn");
lst.add("Uranus");
lst.add("Neptune");
lst.add("Pluto");
```

The first parameter indicates how many list items will be visible at one time. Clicking on an item that isn't selected selects it. Clicking on an item that is already selected deselects it. The second parameter in the constructor indicates whether a list permits multiple selections.

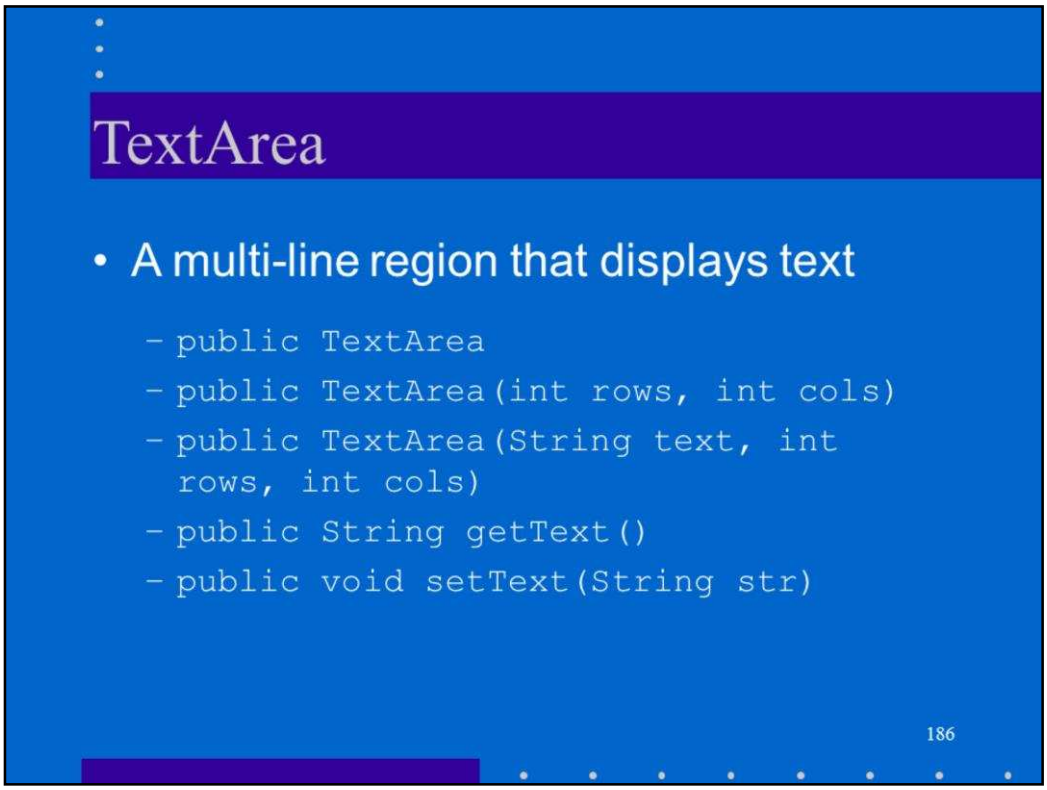

Scrollbar
• a means for allowing a user to select from a range of values
– public Scrollbar
– public Scrollbar(int orientation)
– public Scrollbar(int orientation, int value, int visible, int minimum, int maximum)
– public int getValue()
185

```
public class Scrollbar
extends  Component
implements  Adjustable
```

The `Scrollbar` class embodies a scroll bar, a familiar user-interface object. A scroll bar provides a convenient means for allowing a user to select from a range of values.

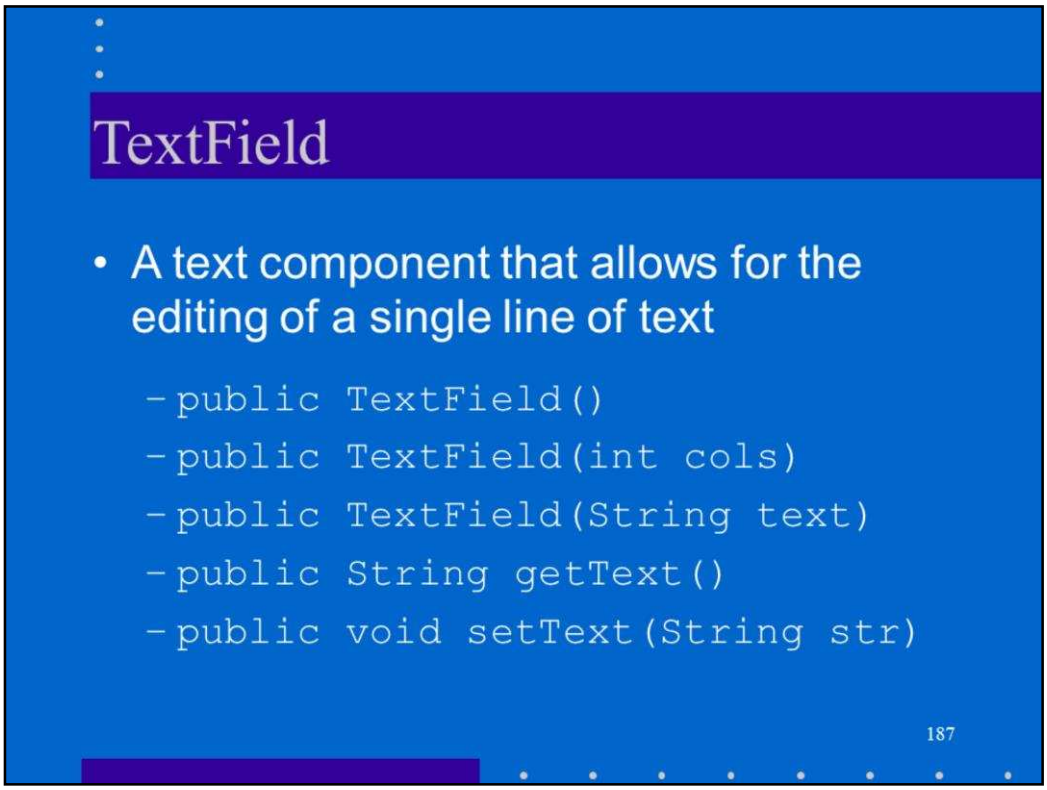
Alternatively, a scroll bar can represent a range of values. For example, if a scroll bar is used for scrolling through text, the width of the "bubble" or "thumb" can represent the amount of text that is visible.

Normally, the user changes the value of the scroll bar by making a gesture with the mouse. For example, the user can drag the scroll bar's bubble up and down, or click in the scroll bar's unit increment or block increment areas. Keyboard gestures can also be mapped to the scroll bar. By convention, the **Page Up** and **Page Down** keys are equivalent to clicking in the scroll bar's block increment and block decrement areas.


TextArea
• A multi-line region that displays text
– public TextArea
– public TextArea(int rows, int cols)
– public TextArea(String text, int rows, int cols)
– public String getText()
– public void setText(String str)
186

```
public class TextArea
extends TextComponent
```

A TextArea object is a multi-line region that displays text. It can be set to allow editing or to be read-only.


TextField
• A text component that allows for the editing of a single line of text
– public TextField()
– public TextField(int cols)
– public TextField(String text)
– public String getText()
– public void setText(String str)
187

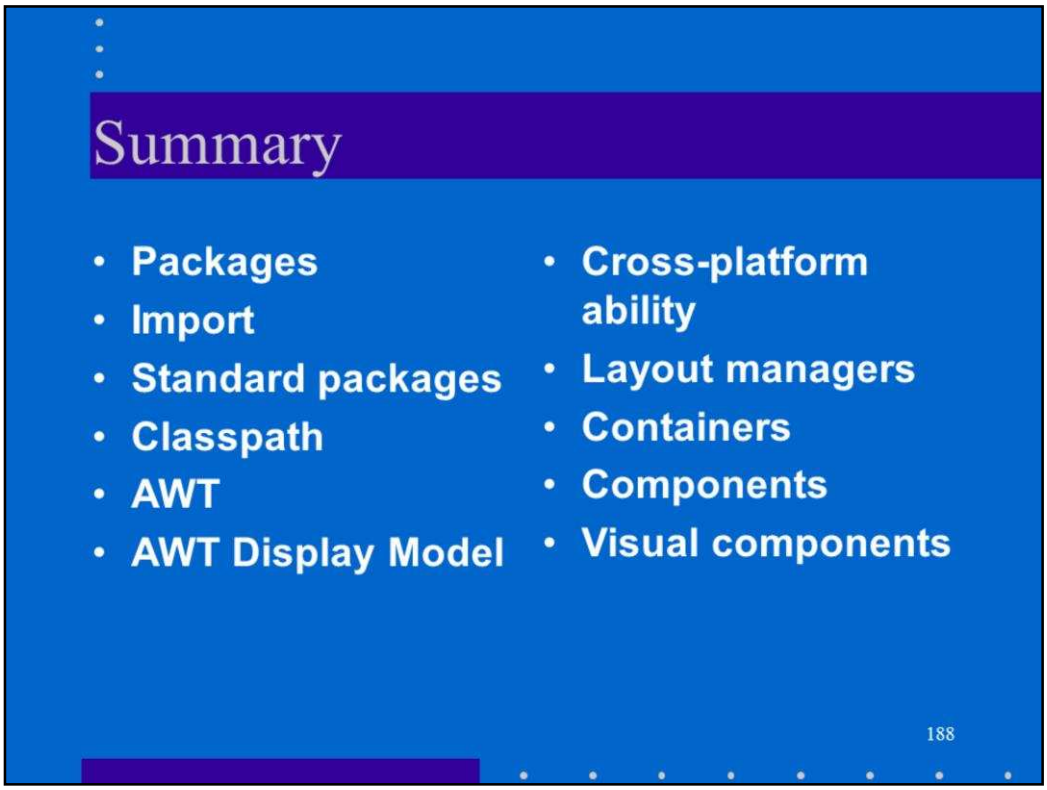
public class TextField
extends

A `TextField` object is a text component that allows for the editing of a single line of text.

Every time the user types a key in the text field, AWT sends two action events to the text field. The first one represents the key press and the second one, the key release. Each action event embodies the state of the system at the time that some action occurred. The properties of an action event indicate which key was pressed, what modifier keys were also pressed, and the time at which the event occurred.

Since the event is an instance of `ActionEvent`, the `TextField` class's `processEvent` method examines the event and passes it along to `processActionEvent`. The latter method redirects the event to any `ActionListener` objects that have registered an interest in action events generated by this text field.

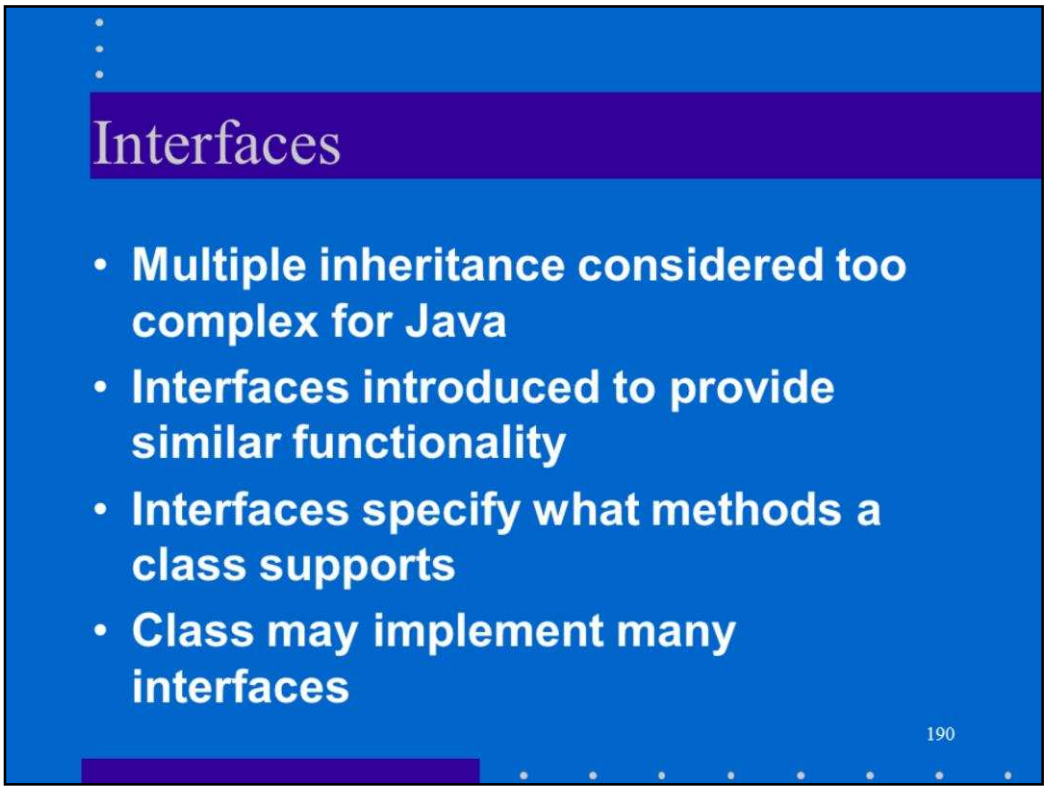
Full details of all the capabilities of AWT can be found in the online Java documentation.



This lecture has explained how Java uses packages. It has also described some of the features of the Java AWT package. So far we have only dealt with the way in which user interface features are added to a window. The next lecture will look at how to deal with user interactions.

Lecture 6

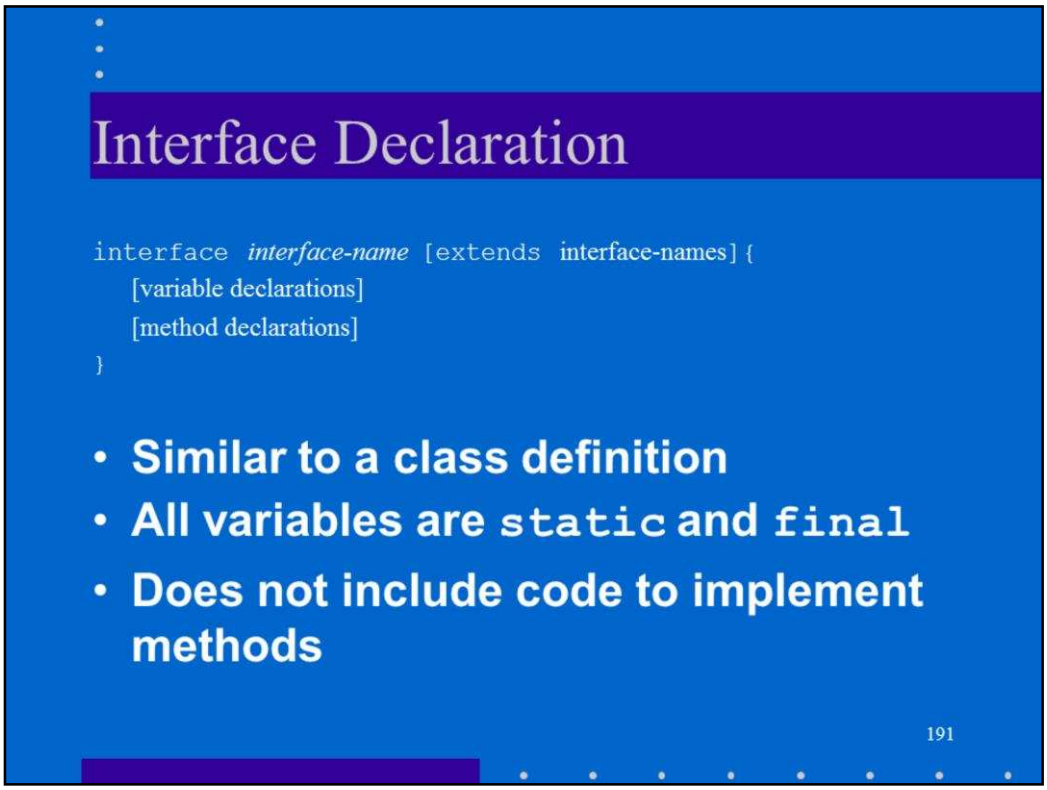
Interfaces, Events and Event Handlers


Interfaces
• Multiple inheritance considered too complex for Java
• Interfaces introduced to provide similar functionality
• Interfaces specify what methods a class supports
• Class may implement many interfaces
190

Interfaces are a mechanism to give Java something which approaches multiple inheritance without the complexity of multiple inheritance.

If a class implements an interface then it guarantees that it will provide code for the methods which form that interface. Those methods can be invoked in full confidence that they exist. If I know a class implements an interface, I know some of the messages I may send it.

Classes may only inherit from one other class but they may implement more than one interface.



An `interface` definition looks similar to a class definition. But all variable definitions in an `interface` are in fact constant definitions and an `interface` may extend more than one `interface`. The method definitions in an `interface` simply define their signatures, they do not contain code.

Interface Example

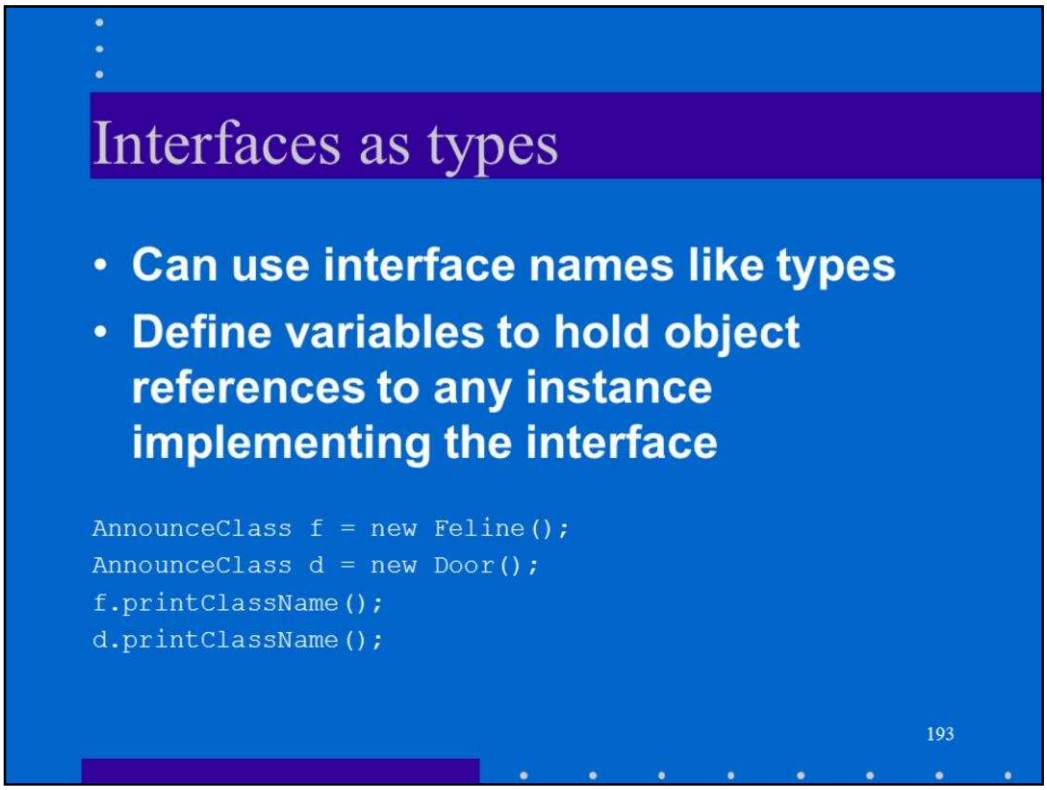
```
interface AnnounceClass{
   void printClassName();
}
class Feline implements AnnounceClass{
   :
   :
   public void printClassName(){
        System.out.println("Feline");
   }
}
class Door implements AnnounceClass{
   :
   :
   public void printClassName(){
        System.out.println("Door");
   }
}
```

192

The `interface AnnounceClass` says that every class that implements it will contain a method called `printClassName()`, which does not expect an argument and does not return anything.

In the example `Feline` and `Door` implement `AnnounceClass`. `Feline` and `Door` are very different classes and it is difficult to imagine that they could ever belong to the same inheritance hierarchy. On the other hand, there are operations we could imagine carrying out on both. If they appeared in a virtual reality simulation then although they are very different, one might expect them both to implement a method called `display()`.

In this case `printClassName` is a method which prints out their class name and both classes can implement this in much the same way.


Interfaces as types
• Can use interface names like types
• Define variables to hold object references to any instance implementing the interface
AnnounceClass f = new Feline();
AnnounceClass d = new Door();
f.printClassName();
d.printClassName();
193

Interface names can be used to define the type of variables which will hold object references to classes. The manner in which they are defined looks identical to declarations which use built in types and user defined classes. The methods declared in the interface definition can then be invoked for these object references. This is particularly useful when the object references are to be stored in an array.

Arrays must have a type declared for them. If a set of objects from different classes all support the same interface, they may be placed in an array whose type corresponds to that interface.

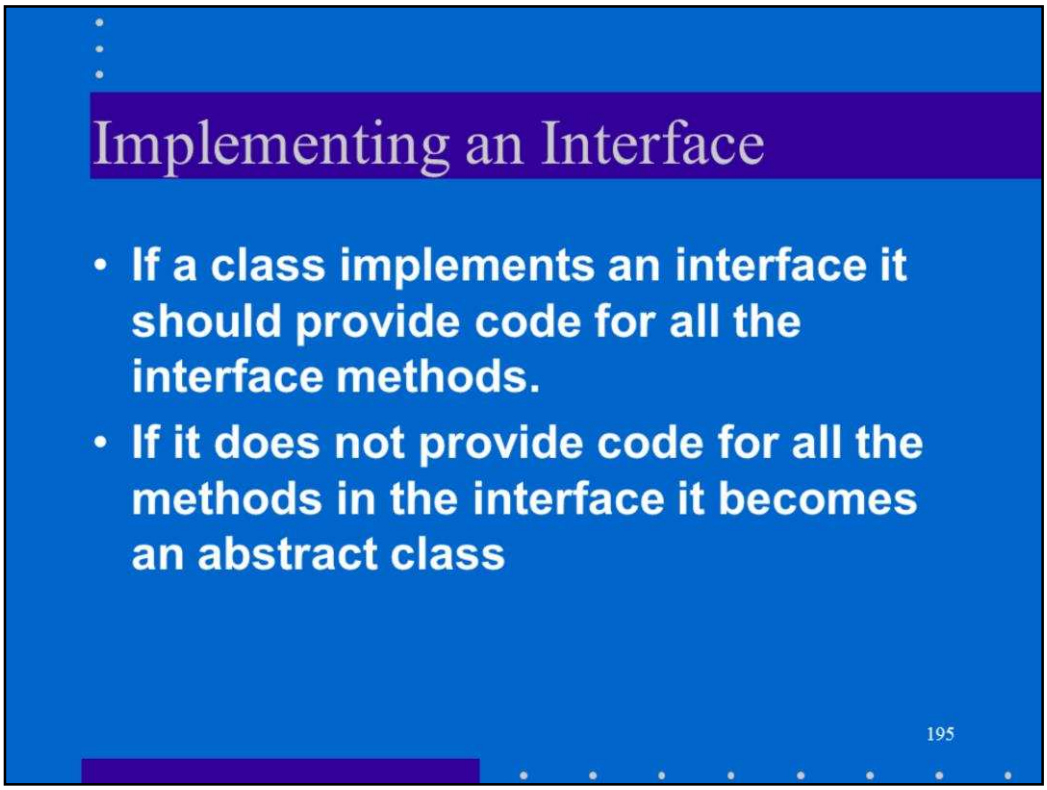
Heterogeneous Arrays

- **Interfaces make it possible to build data structures where members come from non-related classes**

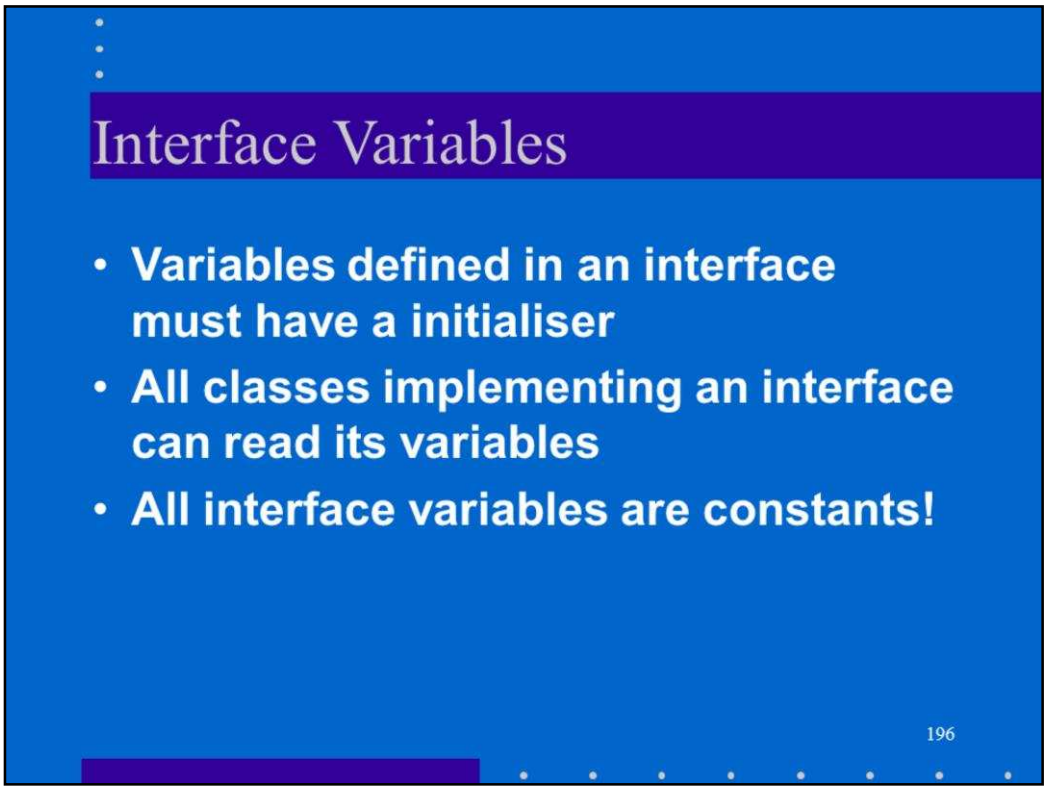
```
class InterfaceArrayTest{
   public static void main(String Args[])
   {
        AnnounceClass[] myList = new AnnounceClass[2];
        myList[0] = new Feline();
        myList[1] = new Door();
        for(int i=0;i<2;i++)myList[i].printClassName();
   }
}
```

194

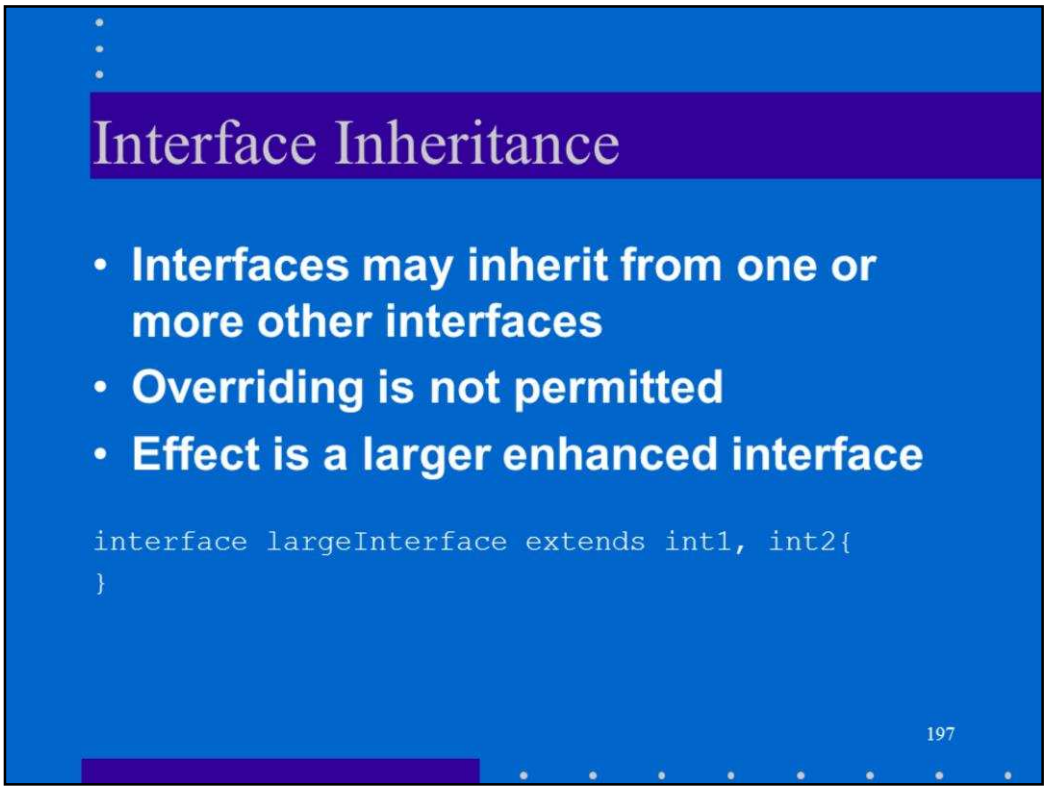
This program declares an array which holds object references to objects which implement the `AnnounceClass` interface. These could be objects which belong to classes which differ widely from each other, however they must implement the methods listed in the `AnnounceClass` interface. In the code shown, references to an object of class Door and an object of class Feline are added to the array. In the `for` loop the method which is common to the two classes, `printClassName()`, is applied to each object referenced by the array.


Implementing an Interface
• If a class implements an interface it should provide code for all the interface methods.
• If it does not provide code for all the methods in the interface it becomes an abstract class
195

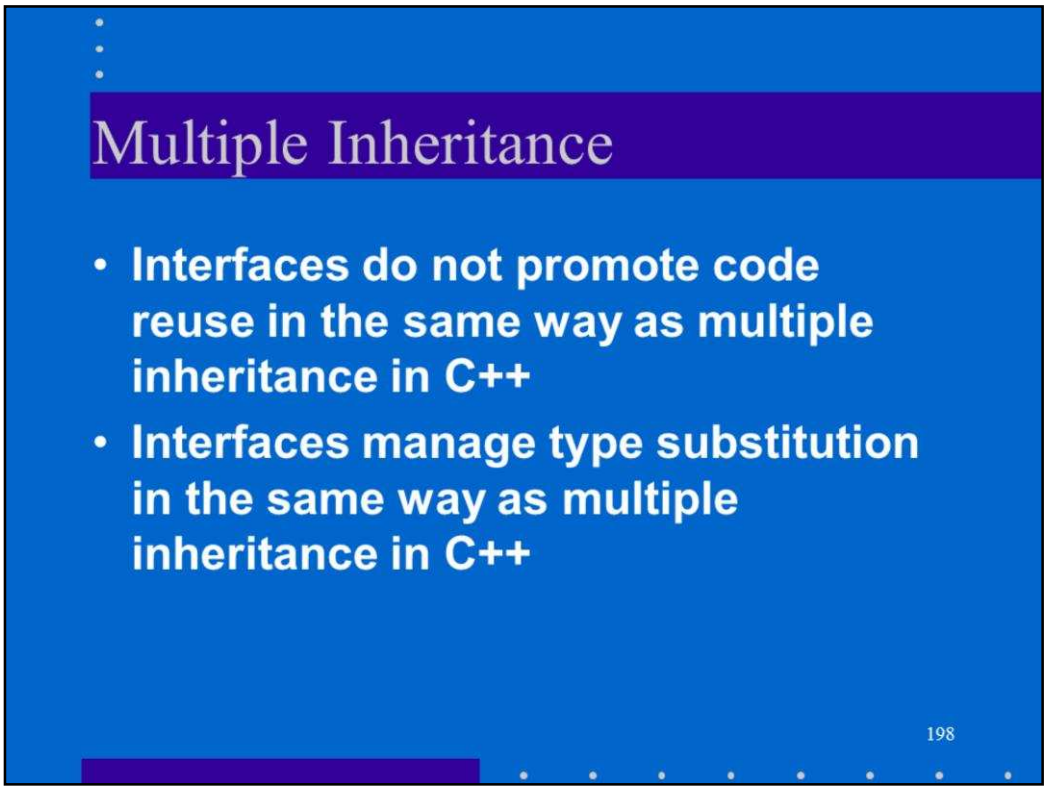
Implementing an interface is a commitment to providing code for the methods named in the interface. A class which fails to provide code for all the methods of the interfaces it implements becomes an abstract class, this means that it cannot be used to create objects.


Interface Variables
• Variables defined in an interface must have a initialiser
• All classes implementing an interface can read its variables
• All interface variables are constants!
196

Variables defined in interfaces become global (to methods implementing the interface) constants! They must be assigned a value. All the methods of all the classes implementing the interface will have access to those values.


Interface Inheritance
• Interfaces may inherit from one or more other interfaces
• Overriding is not permitted
• Effect is a larger enhanced interface
interface largeInterface extends int1, int2{
}
197

An interface may extend one or more other interfaces. But it may not override any methods or redefine any of the variables. The effect is to produce a large interface with the union of all the variables and methods supported by the extended interface plus any new ones introduced by the inheriting interface.


Multiple Inheritance
• Interfaces do not promote code reuse in the same way as multiple inheritance in C++
• Interfaces manage type substitution in the same way as multiple inheritance in C++
198

The C++ multiple inheritance method allows a subclass to inherit methods and data from the all its superclasses.

A Java class inherits methods and data from its one superclass. It must provide code to implement the methods named in any interfaces it implements.

In this respect Java is different to C++.

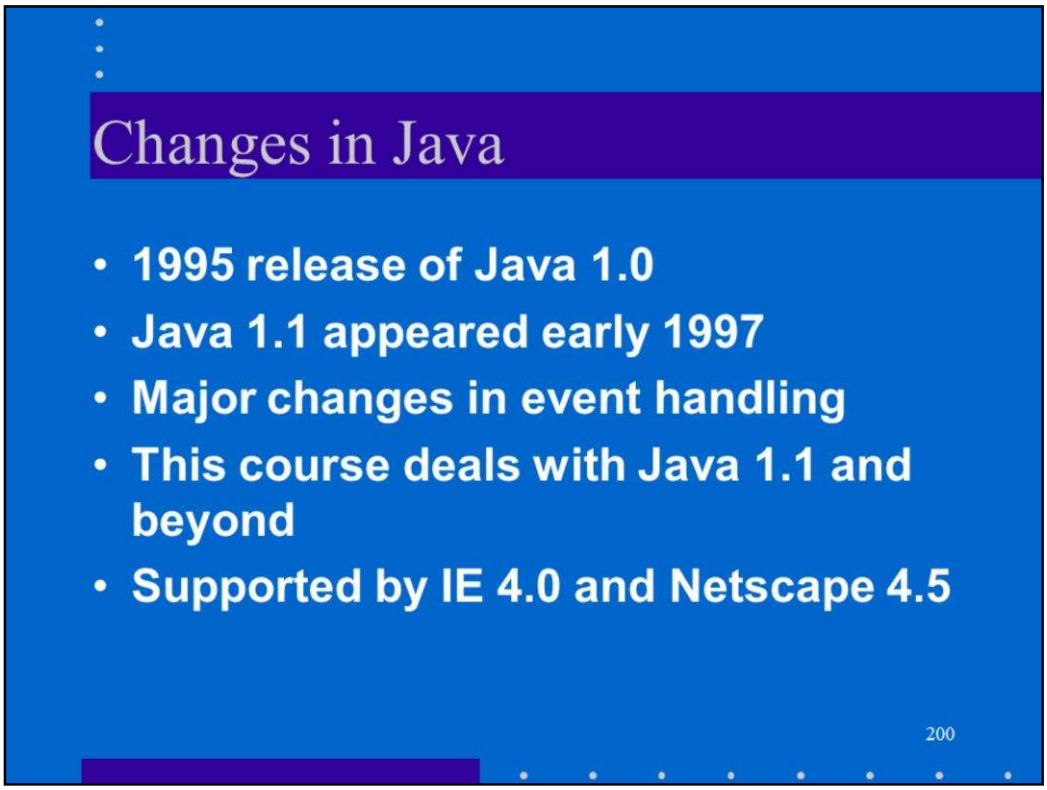
In another sense the use of interfaces is similar to multiple inheritance in C++. The creator of C++ identifies type substitution as the major reason for implementing multiple inheritance. In C++ if class X inherits from class A and class B then an object of class X can be used wherever an object conforming to type A or type B is required.

In Java if a class X inherits from class A and implements an interface I then an object of class X can be used wherever an object conforming to type A or type I is required.

In this respect Java is similar to C++.

The statement that interfaces are a substitute for multiple inheritance is partially true. They permit type substitution in a way which is similar to multiple inheritance but they don't promote the same level of code reuse.

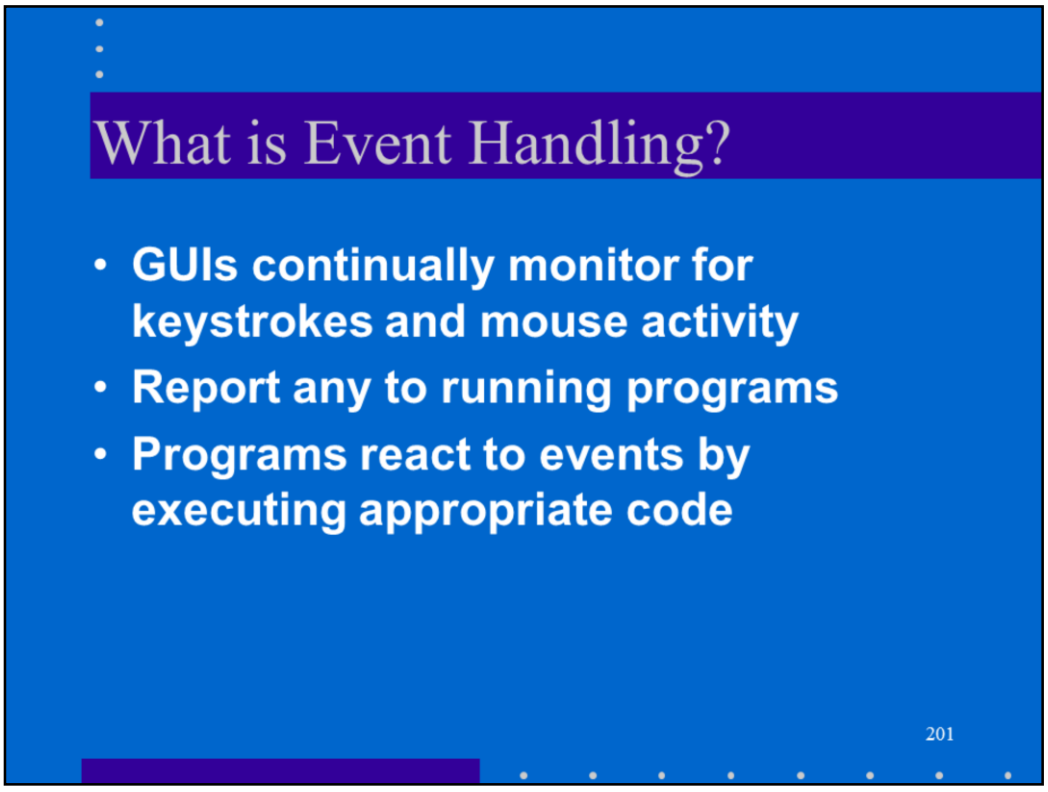
Last week's lecture showed how to add visual components to an applet. The same techniques could be used by an application if the application created a window to hold the components. Whilst the examples shown responded visually to mouse movements etc. they were incomplete in the sense that none of the components communicated with the application. Visual components in Java raise *events* in response to user actions. A program which attempts to respond to user actions must implement an *event handler* to respond to these events.


Changes in Java
• 1995 release of Java 1.0
• Java 1.1 appeared early 1997
• Major changes in event handling
• This course deals with Java 1.1 and beyond
• Supported by IE 4.0 and Netscape 4.5
200

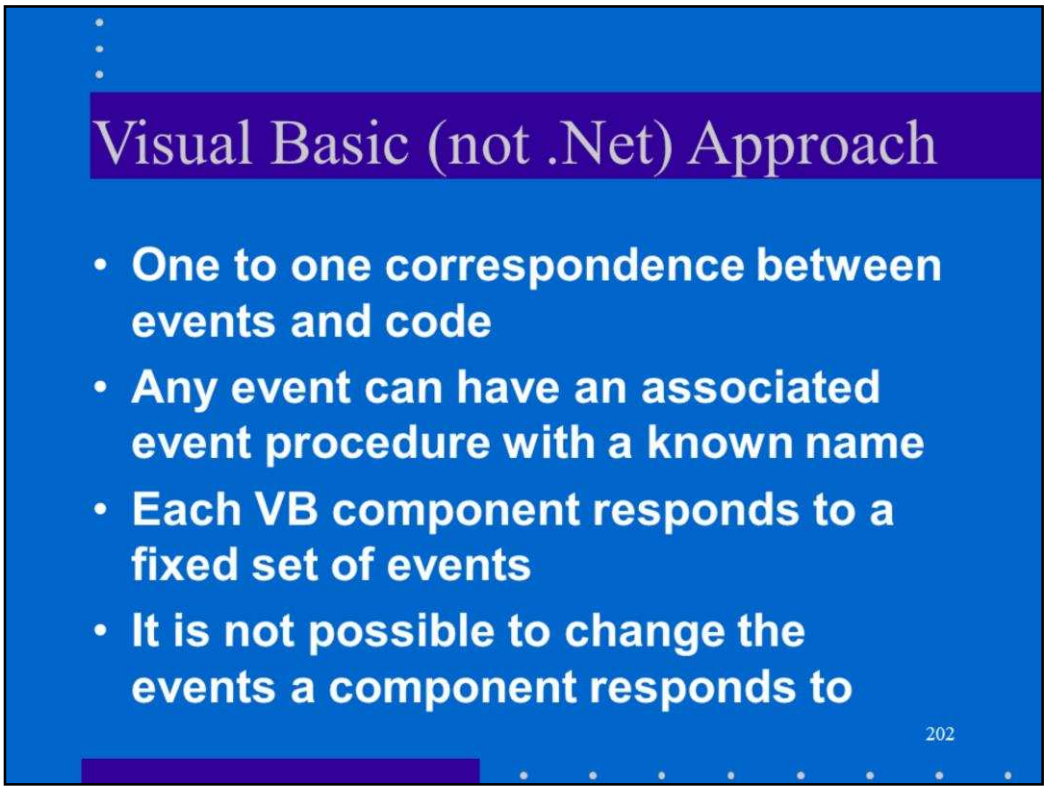
The area of event handling is one which has seen the most changes in the transition from version 1.0 of Java to version 1.1. The 1995 version of Java was more or less 'proof of concept'. With Java 1.1 in 1997 attempts were made to make Java into an 'industrial strength' tool. The event handling mechanism in Java 1.0 was clumsy and inefficient, the version in Java 1.1 is a great improvement.

The world is slowly catching up with the move from Java 1.0 to Java 1.1. Many of the textbooks still available describe Java 1.0. Java 1.1 compilers will allow you to compile Java 1.0 code but will indicate that you have used a 'deprecated' feature. Java 1.1 applets are only supported by the Microsoft Internet Explorer 4 browser and Netscape 4.5. It is unwise, therefore, to put Java applets which use the event handling mechanism described in this lecture into Web pages which will be accessed by people with older versions of browsers.

Java 1.4 has recently been released and is the standard version for standalone applications (written in Jan 2001. 2002). However, the latest browsers do not yet support 1.2 fully so it will be some time before there is support for 1.4 applets.

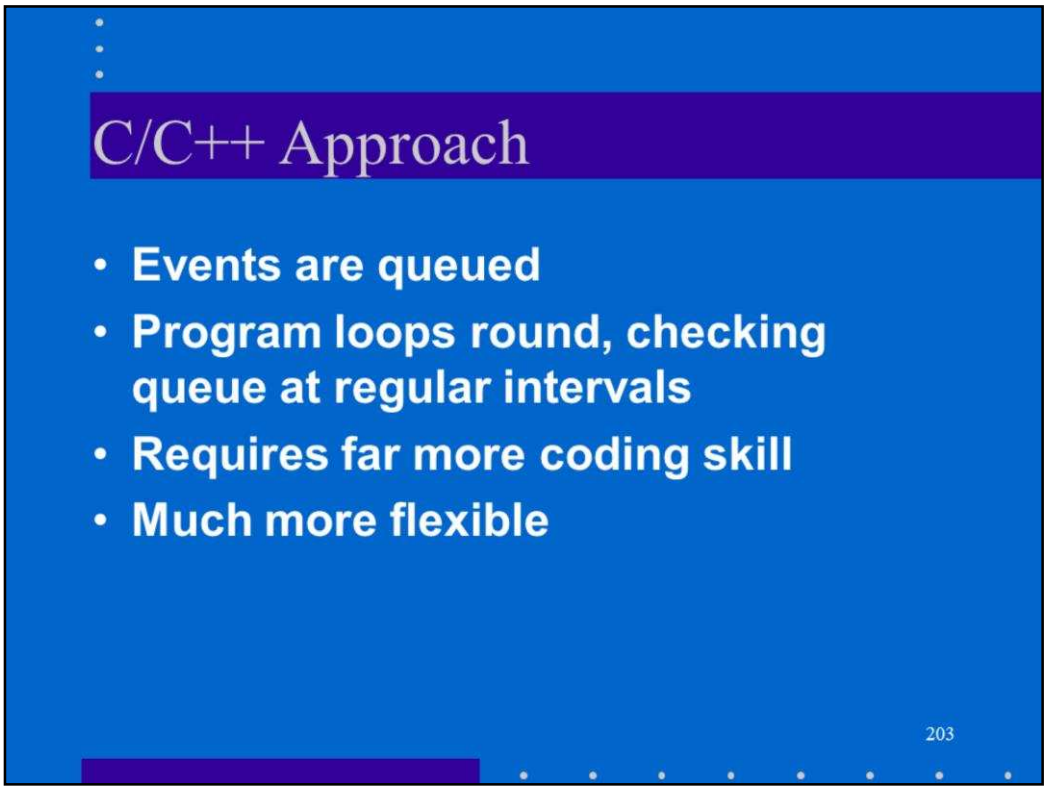

What is Event Handling?
• GUIs continually monitor for keystrokes and mouse activity
• Report any to running programs
• Programs react to events by executing appropriate code
201

Graphical User Interfaces (GUIs) continually check for user input. This input is normally in the form of a keyboard entry or a mouse movement. If any input is detected, then the program that created the GUI is informed. The nature of the way in which the program is informed is through raising an *event*. Each program will have a queue of events associated with it. Programs may choose to ignore events or choose not to examine the event queue for some time. Well behaved programs will always look at the queue sooner or later and then will decide what actions to take for each event in the queue.

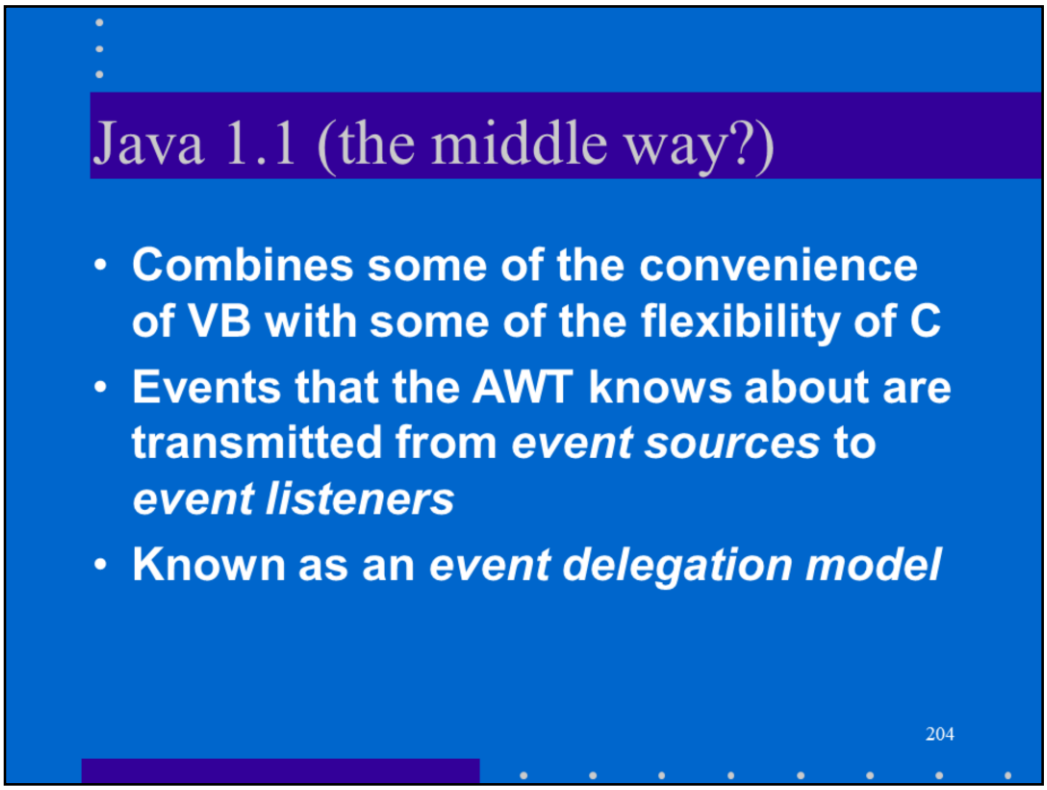

Visual Basic (not .Net) Approach
• One to one correspondence between events and code
• Any event can have an associated event procedure with a known name
• Each VB component responds to a fixed set of events
• It is not possible to change the events a component responds to
202

Visual Basic makes event handling a very transparent process. There is a procedure for every possible event known to the language. So if a button MyButton exists on the screen then there must be a procedure MyButton_Click associated with it. The VB programmer simply has to decide what code to write into this procedure in order to handle the event.

This is very straightforward but also very limited. Only components provided in the language can appear on the screen. Each component responds to a fixed set of events, it isn't possible to invent new ones.


C/C++ Approach
• Events are queued
• Program loops round, checking queue at regular intervals
• Requires far more coding skill
• Much more flexible
203

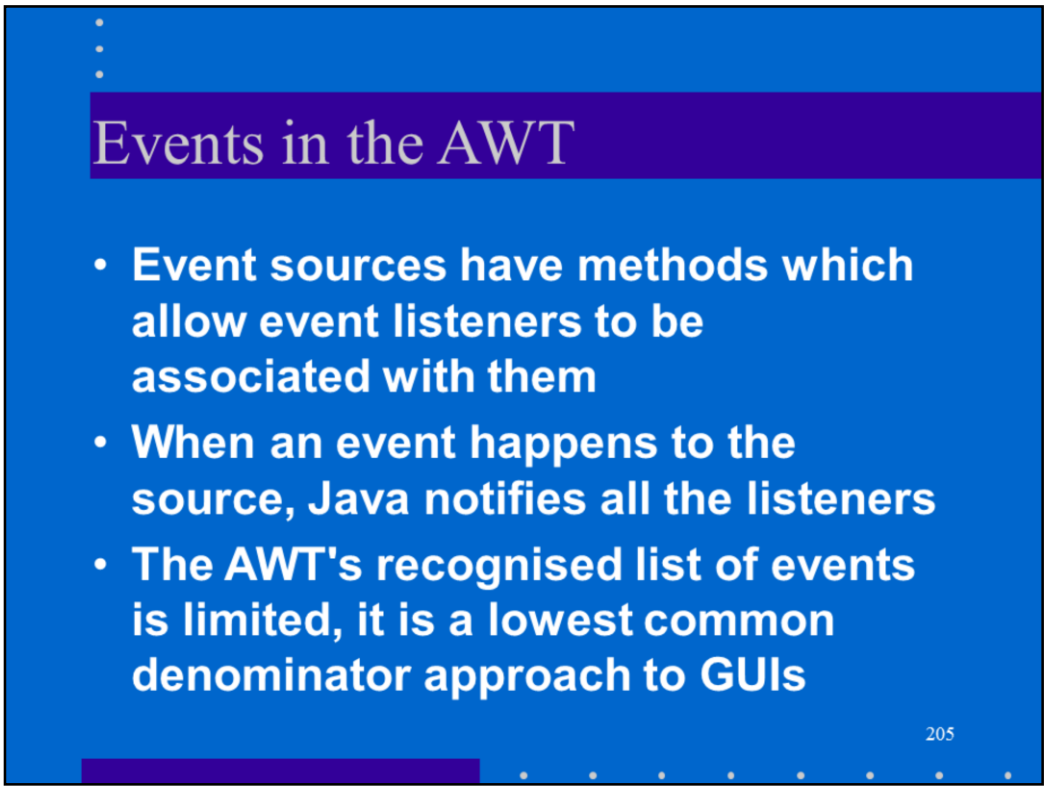
The C/C++ approach is to examine the event queue directly. At the heart of most GUI programs written in C or C++ is a loop which gets events from the event queue and then uses a switch statement to decide what to do about them. Programmers have to remember to make sure control returns to this loop periodically. As with most things in C and C++, this gives the programmer complete control over the way that events are handled. The disadvantage is that the programmer has to take much more care that the correct action is taken for each event. It only takes a small slip in the coding and every action is executed for every event!


Java 1.1 (the middle way?)
• Combines some of the convenience of VB with some of the flexibility of C
• Events that the AWT knows about are transmitted from event sources to event listeners
• Known as an event delegation model
204

Java tries to steer a middle course between these two approaches. The programmer is given considerable help to ensure that the intended action is executed for each event. On the other hand the approach is much more flexible than that found in Visual Basic.

Visual components in Java are regarded as event sources. Event handlers can have names which are chosen by the programmer, rather than the system, and an event handler may, if necessary, handle events from multiple event sources. The Java mechanism enables a connection to be made between an event source and the event handler the programmer has selected to deal with events from that source. Both event sources and event handlers are, of course, objects.

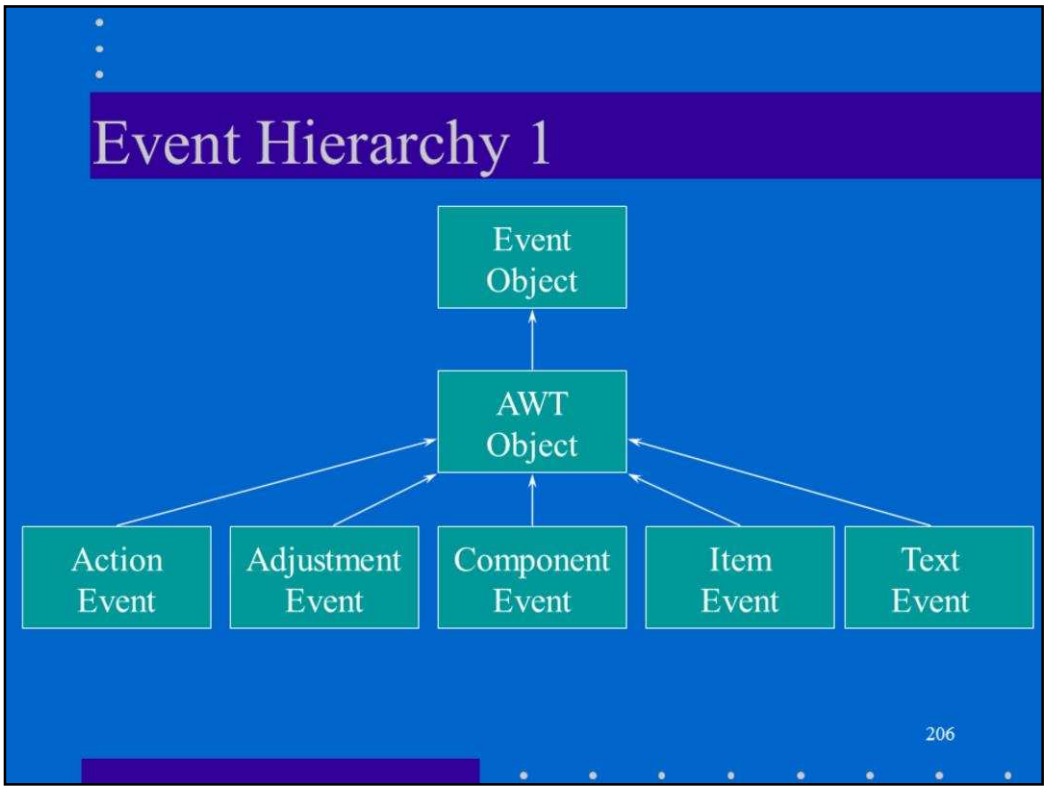
The result is a mechanism which is more flexible than VB and easier than C++. Or looked at another way, less flexible than C++ and more difficult to master than VB!


Events in the AWT
• Event sources have methods which allow event listeners to be associated with them
• When an event happens to the source, Java notifies all the listeners
• The AWT's recognised list of events is limited, it is a lowest common denominator approach to GUIs
205

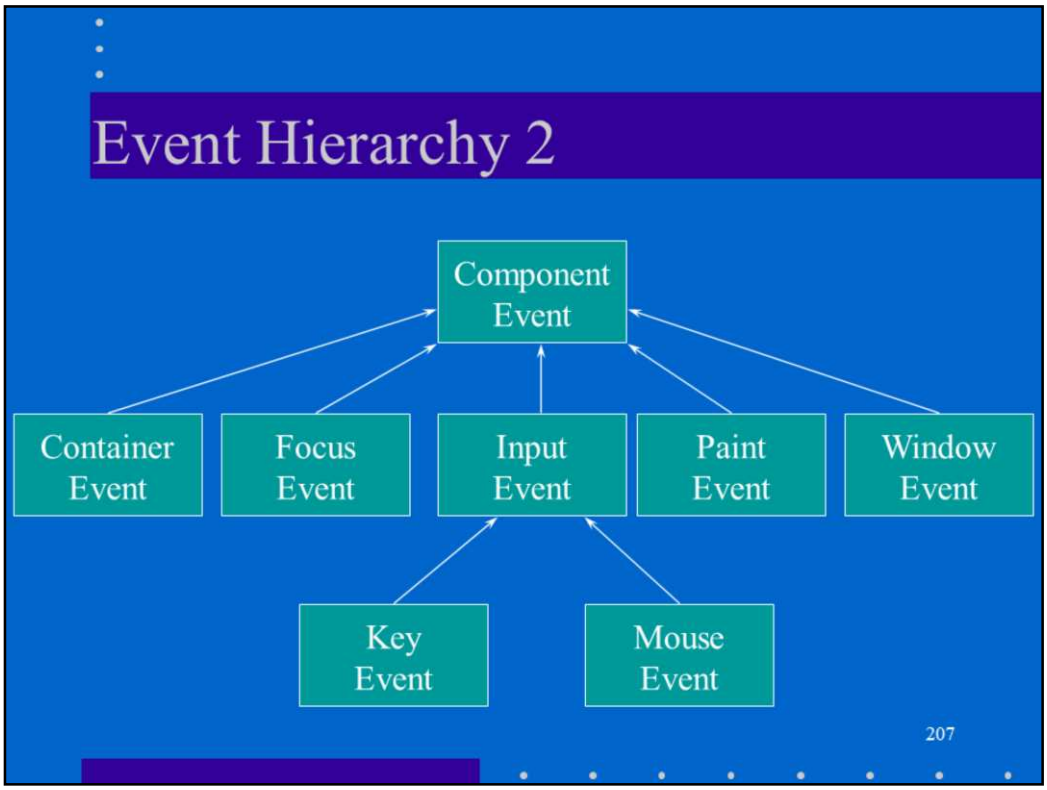
All event sources are components and inherit from the `Component` class the ability to have event listener objects associated with them. The effect of registering an object as an event listener is to say to the Java run-time system, "when an event happens to this component tell this object by calling this method".

More than one listener can be associated with a component. When an event takes place which involves the component then all the associated listeners are notified. Listeners which are not associated with that component are not notified.

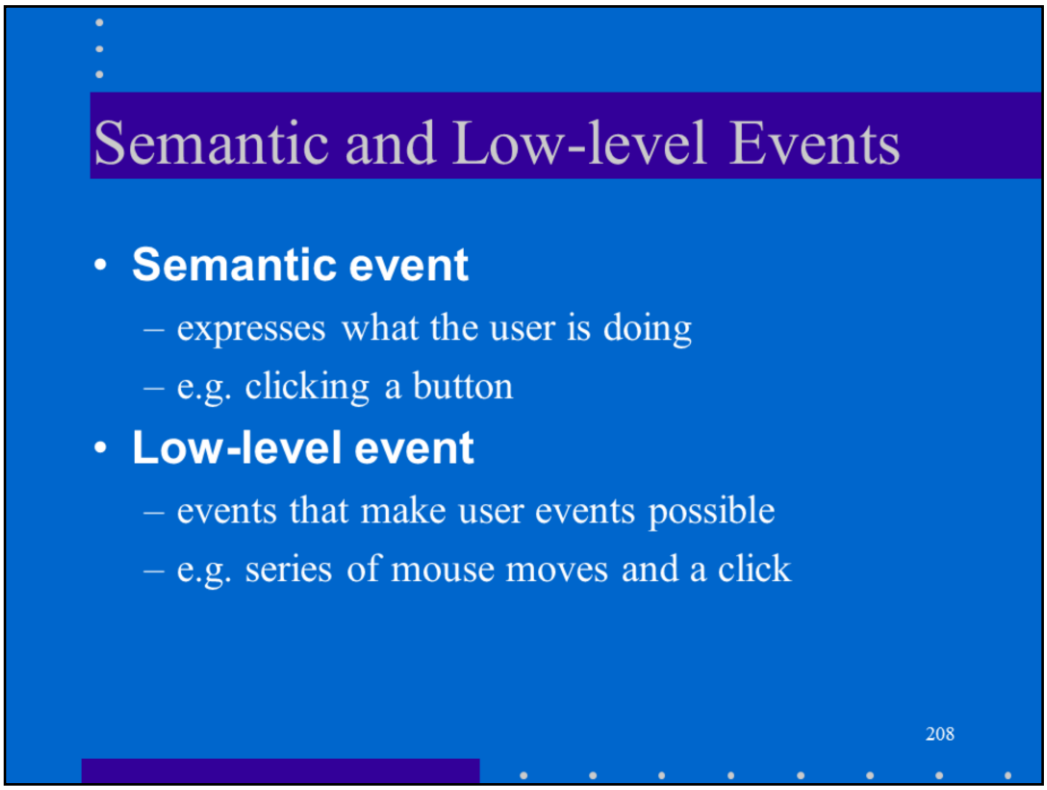
As with the visual components supported by the AWT, the events recognised are limited to those common to the most popular GUI environments. Event types only found in one particular GUI system are unlikely to be supported by the AWT as this would hinder portability.



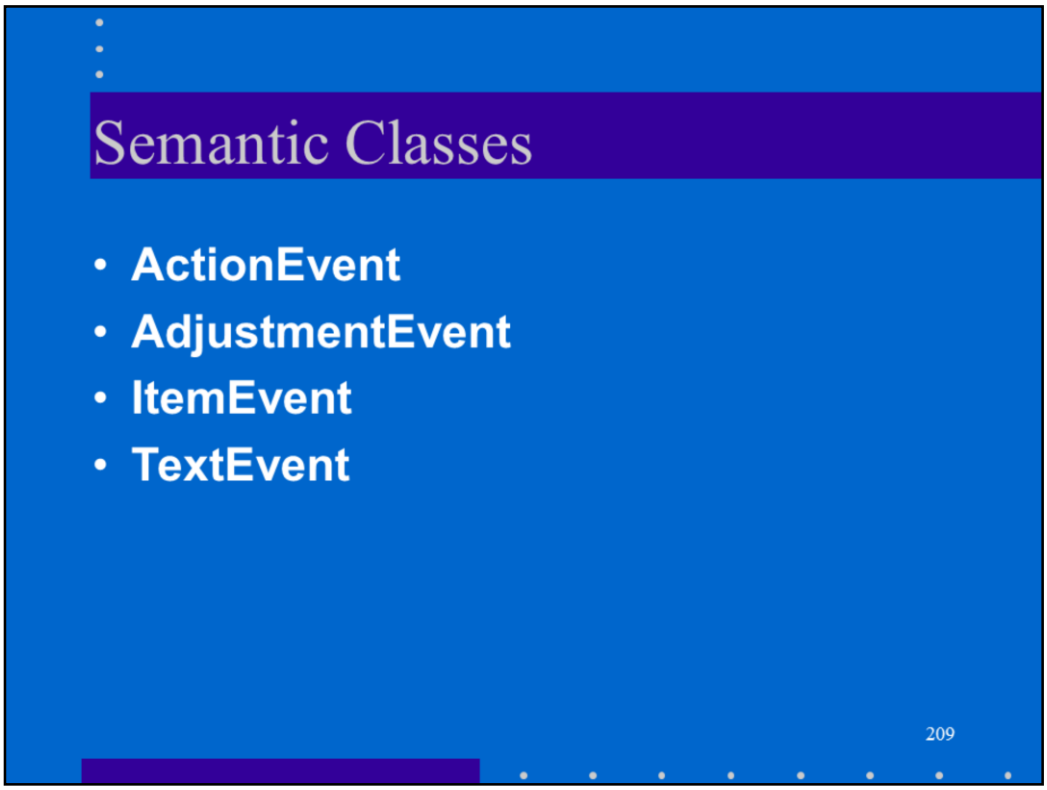
Events in Java are organised in an inheritance hierarchy. It is possible to add new classes to this inheritance hierarchy and hence deal with new types of events.



Classes shown here represent all the objects which may appear on the event queue, however not all events are of interest to programmers. In particular a `PaintEvent` is not handled in the same way as the other events.


Semantic and Low-level Events
• Semantic event
– expresses what the user is doing
– e.g. clicking a button
• Low-level event
– events that make user events possible
– e.g. series of mouse moves and a click
208

Java distinguishes between semantic events and low-level events. For most applications semantic events, which represent what the user does in terms of manipulating buttons, slider bars etc. are sufficient. In a graphics program, however, low-level events would become significant. A semantic event will consist of a sequence of low-level events.

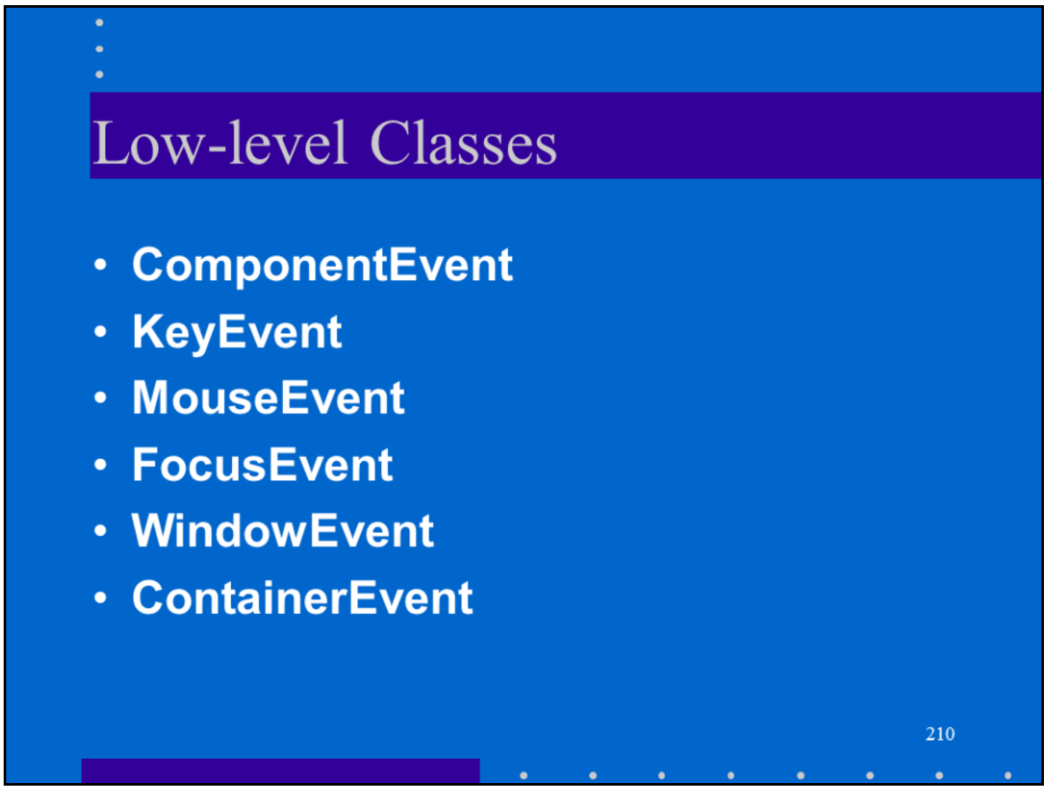


An `ActionEvent` occurs when a user clicks a button, makes a menu selection, double clicks on a list box item or presses [`Enter`] in a text field.

An `AdjustmentEvent` occurs when a user adjusts a scroll bar.

An `ItemEvent` occurs when a user selects something from a list or ticks a checkbox.

A `TextEvent` occurs when a user enters text into a text box or a text area.



A `ComponentEvent` occurs when a component is resized, moved, shown or hidden.

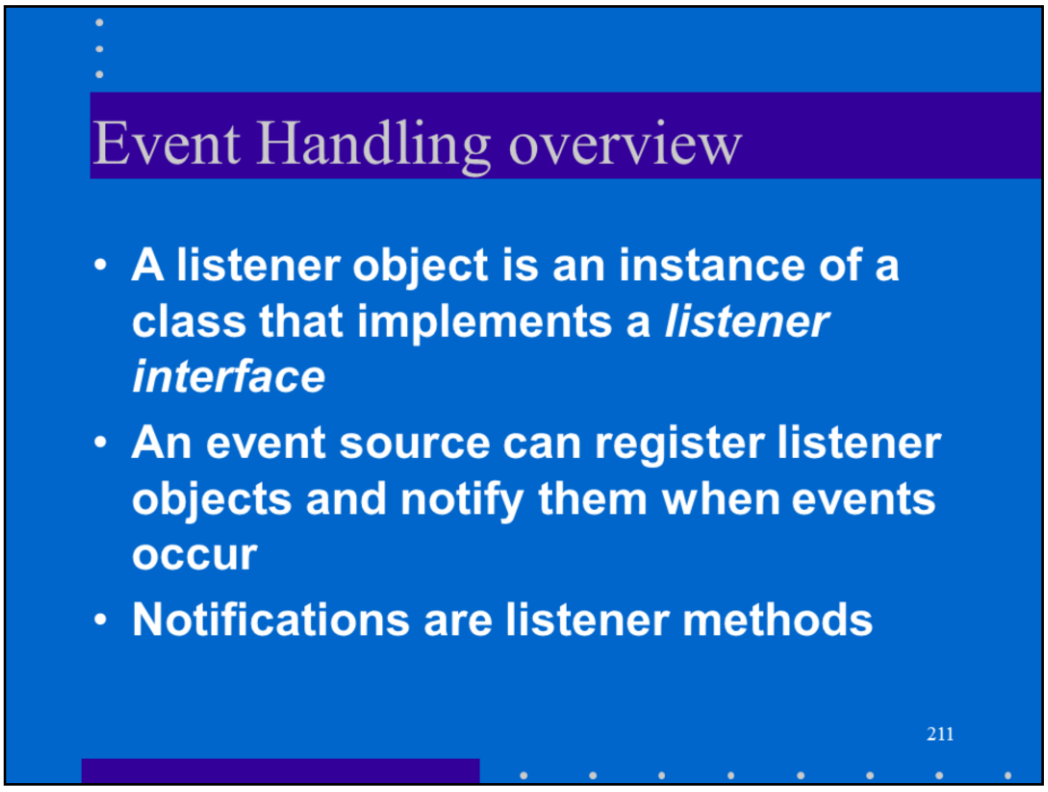
A `KeyEvent` occurs every time a key is pressed or released.

A `MouseEvent` occurs whenever a mouse button is depressed or released and whenever the mouse is moved.

A `FocusEvent` occurs whenever a component gets focus.

A `WindowEvent` occurs whenever a window is activated, deactivated, iconified, deiconified or closed.

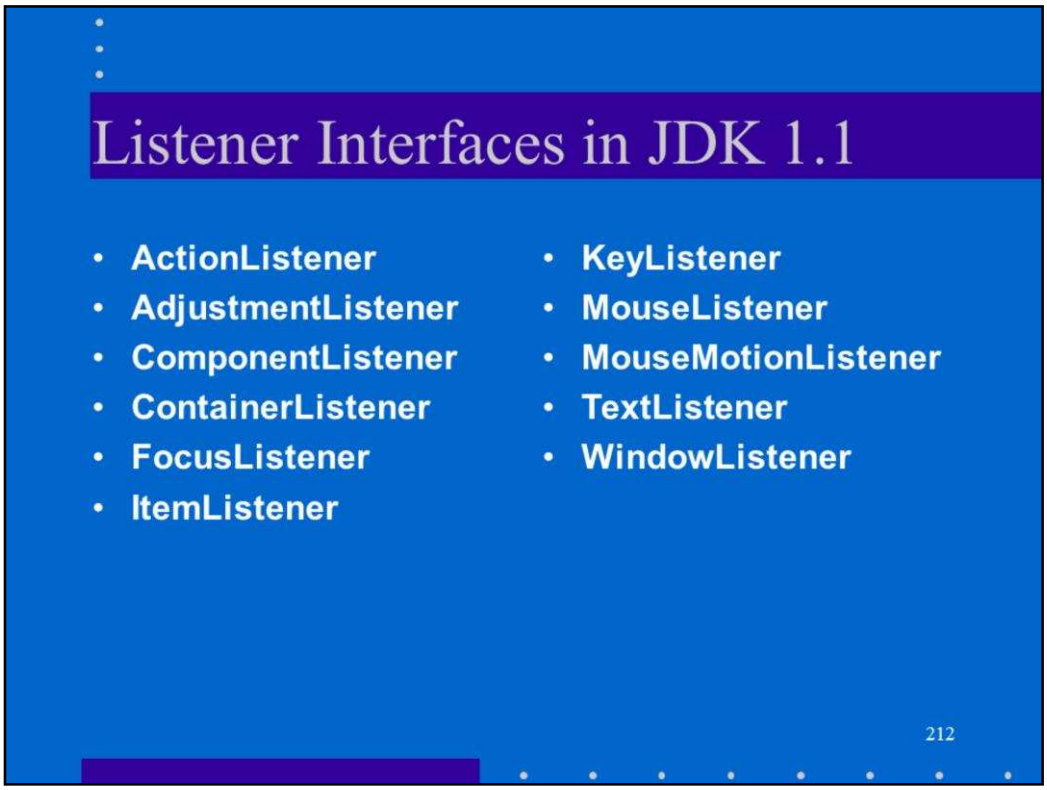
A `ContainerEvent` occurs whenever a component is added or removed from a container.


Event Handling overview
• A listener object is an instance of a class that implements a *listener interface*
• An event source can register listener objects and notify them when events occur
• Notifications are listener methods
211

Objects that can listen for events must be instantiated from classes that implement listener interfaces. The listener interfaces are defined in the JDK.

Notification takes place automatically when an event occurs to an event source. The run time system implements this mechanism.

Since all listeners implement a listener interface, the system can rely on the fact that listener objects will implement certain named methods. Notification of an event is therefore the invocation of a method.



JDK1.1 defines 11 sorts of listener interfaces. These are in the package java.awt.event.
An `ActionListener` listens for action such as a button press.

An `AdjustmentListener` listens for value adjustments that occur when the value of a scrollbar is changed.

A `ComponentListener` listens for events such as the moving, resizing or hiding of a component.

A `ContainerListener` listens for events involving the addition or removal of a component from a container.

A `FocusListener` listens to see if a text box or a text field can receive key strokes.

An `ItemListener` listens to see if the state of an item has changed, e.g. if a check box has been ticked.

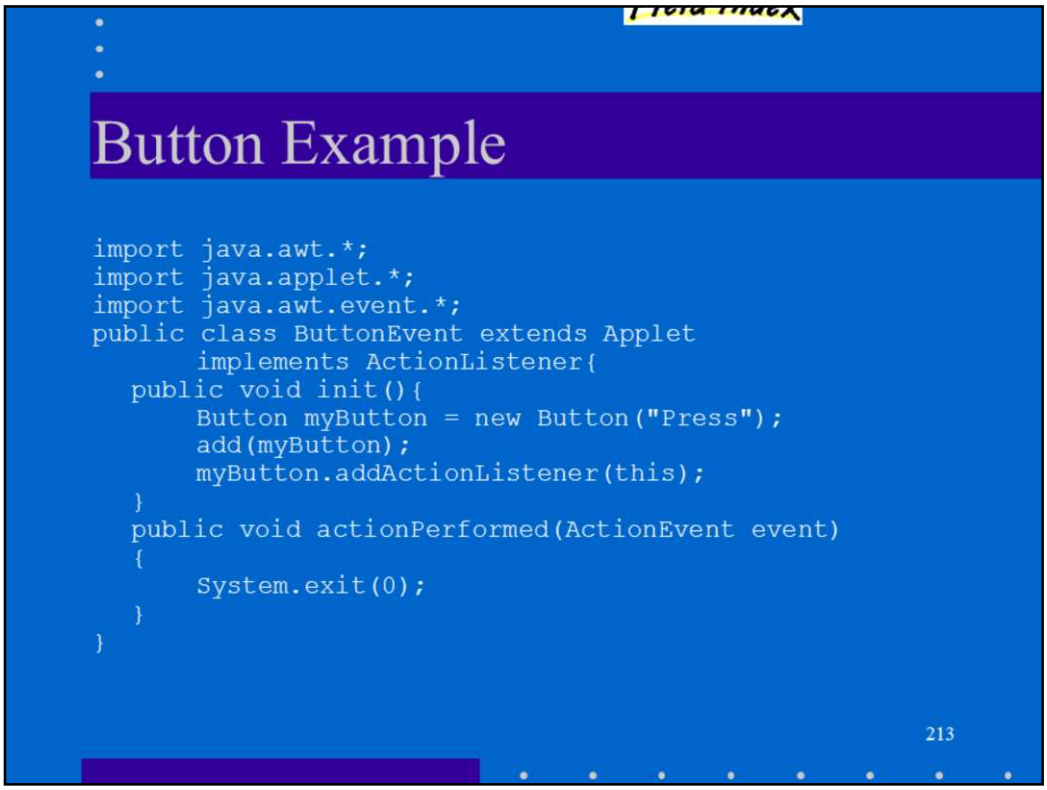
A `KeyListener` listens for keyboard presses.

A `MouseListener` listens for mouse clicks.

A `MouseMotionListener` listens for mouse movements.

A `TextListener` listens for changes in text values.

A `WindowListener` listens for window actions e.g. window closed.



This applet is its own listener. It is an applet because it extends class `Applet` and an `ActionListener` because it implements the `ActionListener` interface.

It creates a button labelled "Press" and then assigns itself (using `this`) as the listener for this button. The method `addActionListener` assigns the listener. All components respond to the methods:

`addComponentListener(ComponentListener)`

`addFocusListener(FocusListener)`

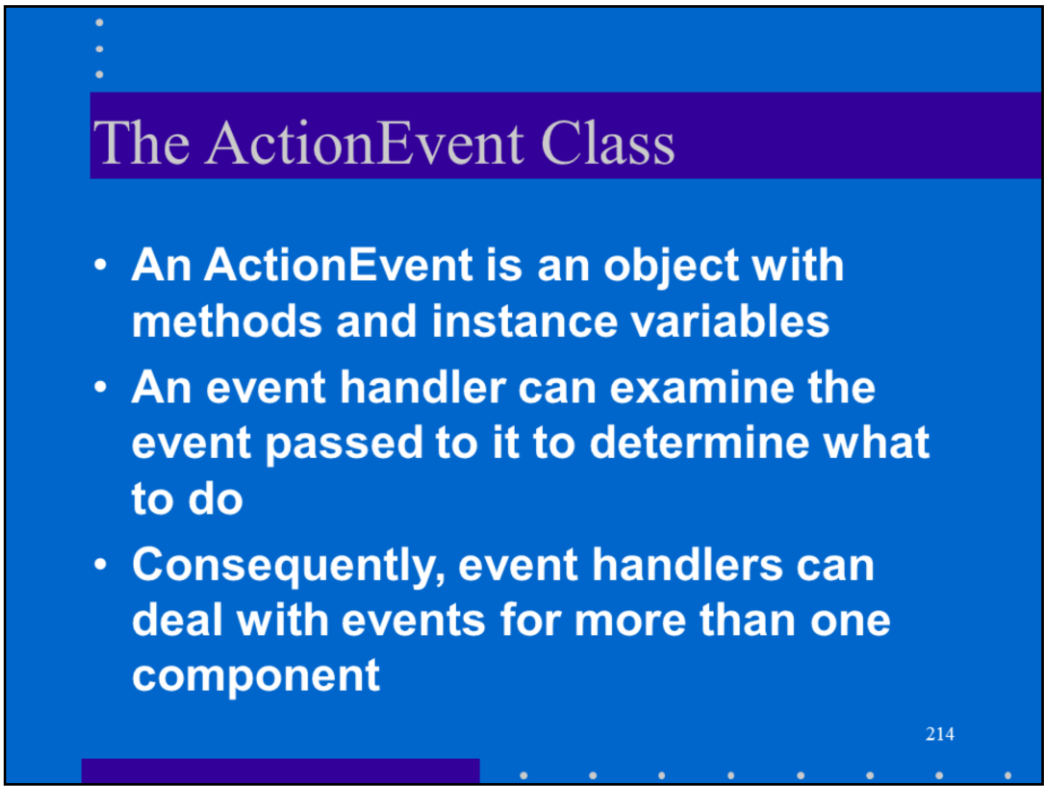
`addKeyListener(KeyListener)`

`addMouseListener(MouseListener)`

`addMouseMotionListener(MouseMotionListener).`

In addition the `Button` class implements `addActionListener`.

Any class that implements the `ActionListener` interface must define a method called `actionPerformed` which accepts an `ActionEvent` parameter. In this case whenever the button is pressed the method `System.exit` is invoked. This closes down the applet.


The ActionEvent Class
• An ActionEvent is an object with methods and instance variables
• An event handler can examine the event passed to it to determine what to do
• Consequently, event handlers can deal with events for more than one component
214

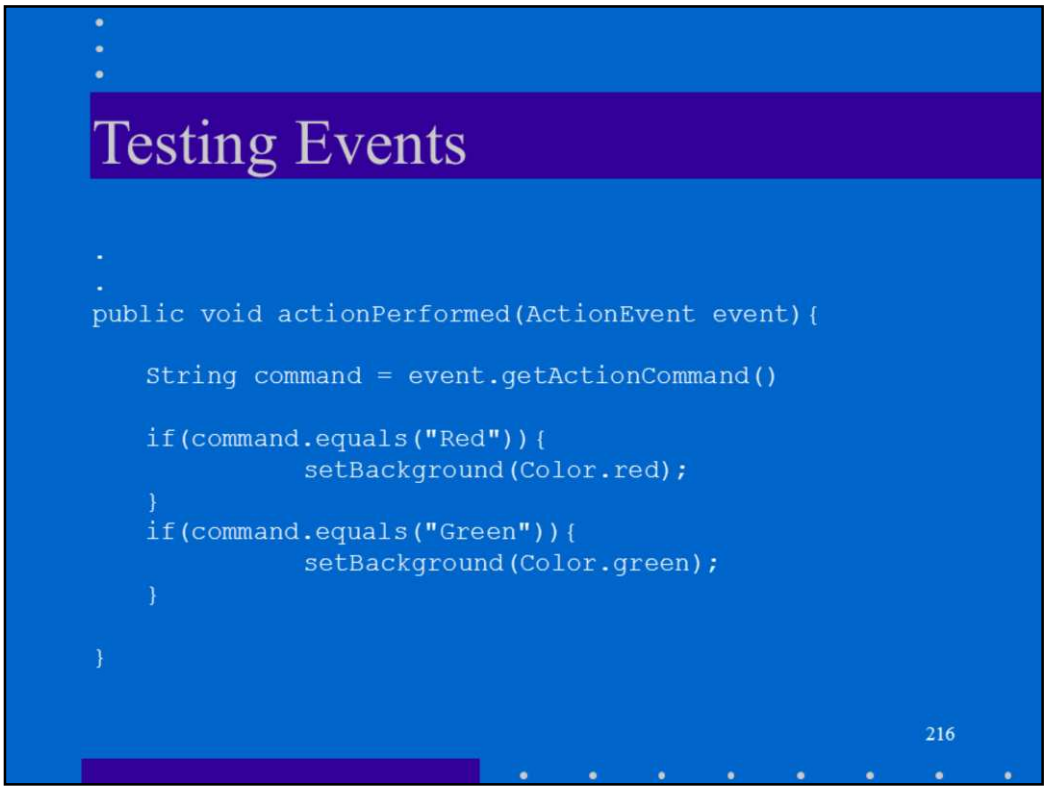
When an event occurs an object is created to describe it. The class of the object that is created will depend on the type of event that has occurred. The event object will respond to method invocation and may have instance variables associated with it. The event object is passed to the relevant event handler. The event handler can use the event's methods and instance variables to find out the details of the event that has occurred. This mechanism can be used as a means of associating a single event handler with more than one component.

Listening on >1 components

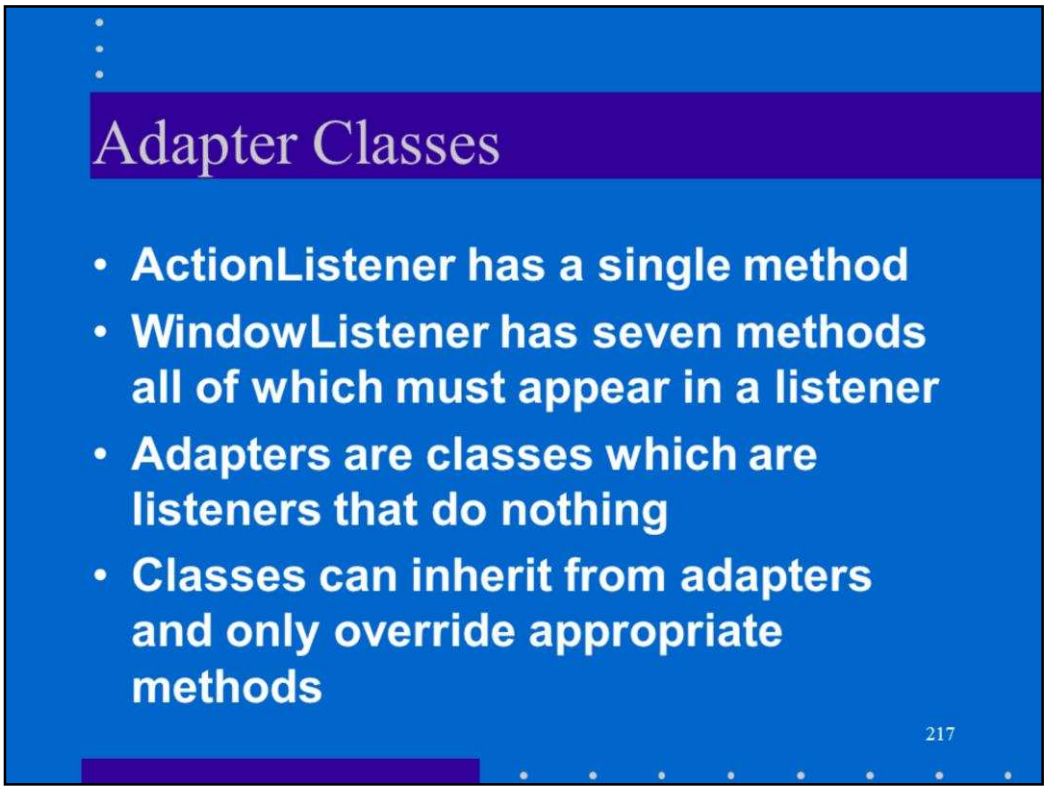
```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class TwoEvent extends Applet
       implements ActionListener{
   public void init(){
        Button redButton = new Button("Red");
        add(redButton);
        redButton.addActionListener(this);
        Button greenButton = new Button("Green");
        add(greenButton);
        greenButton.addActionListener(this);
   }
.
.
```

215

In this code two buttons labelled "Red" and "Green" are created and added to the screen. The applet which implements the `ActionListener` interface nominates itself as the listener for events associated with either of these buttons.

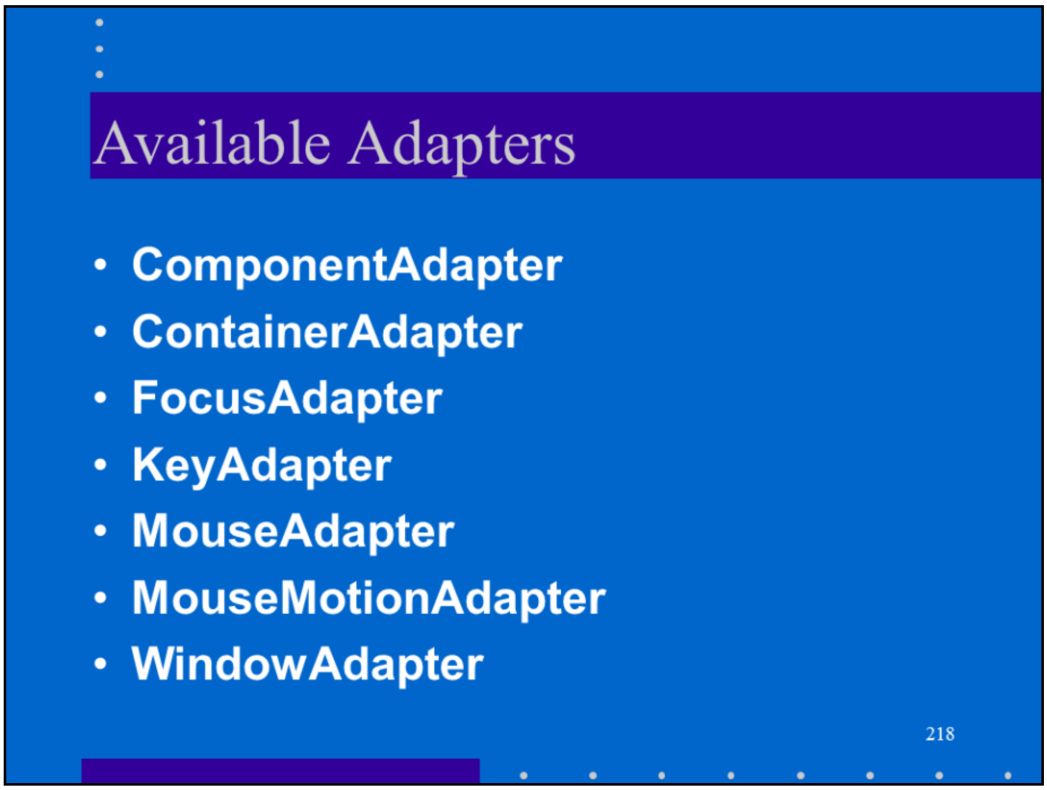


Since the applet implements the `ActionListener` interface it must contain code for all the `ActionListener` methods. There is only one, `actionPerformed`. This method will be called when either the red or the green button is pressed. Clearly it needs to know which one it is dealing with. An `ActionEvent` which is passed to actionPerformed responds to the message `getActionCommand`. When a button event has occurred `getActionCommand` will return the text of the button which generated the event. This will be either "Red" or "Green". The applet sets its background colour appropriately.

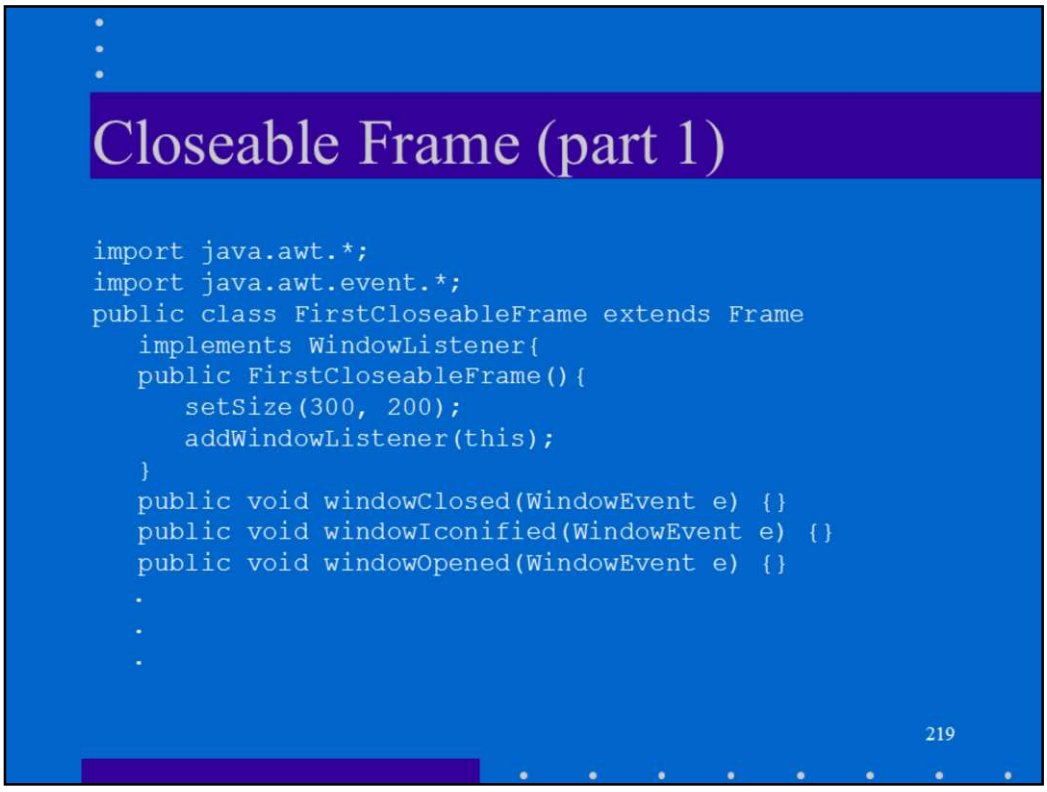

Adapter Classes
• ActionListener has a single method
• WindowListener has seven methods all of which must appear in a listener
• Adapters are classes which are listeners that do nothing
• Classes can inherit from adapters and only override appropriate methods
217

There is only one `ActionListener` method: `actionPerformed`. Some of the other listener interfaces contain many more methods. For example, `WindowListener` contains seven. A class which implements the `WindowListener` interface must provide code for all seven methods. If the class is only designed to respond to a small subset of window events then the majority of the methods will not contain any statements.

To provide a tidier way of specifying listeners, Java supplies adapters. Adapters are classes which implement listener interfaces as "do nothing" methods. Classes which wish to be listeners then simply extend the appropriate adapter and override the methods for any events they wish to handle.



These are the adapters Java supports. Adapters are only supplied for listener interfaces which contain more than one method.



The example shown demonstrates how an application may make use of the Java visual components. An applet can place components into a space provided by the browser it is running in it. An application must create a window in which to place the components. The basic window in Java is implemented as class `Frame`. `FirstCloseableFrame` extends the `Frame` class. Its default constructor sets the size of the window and makes the class a listener for the window it creates.

Since `FirstCloseableFrame` implements the `WindowListener` interface it must provide code for all of the seven `WindowListener` methods. In this case we are only interested in doing anything when the close button on the window is pressed, so six of the methods are superfluous and contain no code. They have to be there or else Java will generate a compilation error.

CloseableFrame (part 2)

```
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}

    public static void main(String[] args){
    Frame f = new FirstCloseableFrame();
       f.show();
    }
}
```

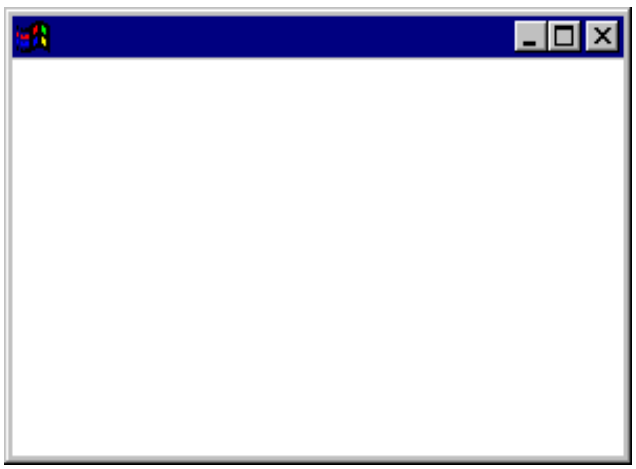
220

The one method which does contain code is `windowClosing` which simply shuts everything down by invoking `System.exit`.

The `main` method creates a frame and tells the frame to show itself. A blank window of the size specified in the constructor will appear.

When the close button is pressed (top rightmost window button with a cross on it in Windows '95) the program will halt and the window will disappear.

As you can see this application contains a lot of unnecessary code!



Using an Adapter

```
import java.awt.*;
import java.awt.event.*;
class WindowCloser extends WindowAdapter{
   public void windowClosing(WindowEvent e){
        System.exit(0);}
}
public class SecondCloseableFrame extends Frame{
   public SecondCloseableFrame(){
        setSize(300, 200);
        addWindowListener(new WindowCloser());
   }
   public static void main(String[] args){
        Frame f = new SecondCloseableFrame();
        f.show();
   }
}
```

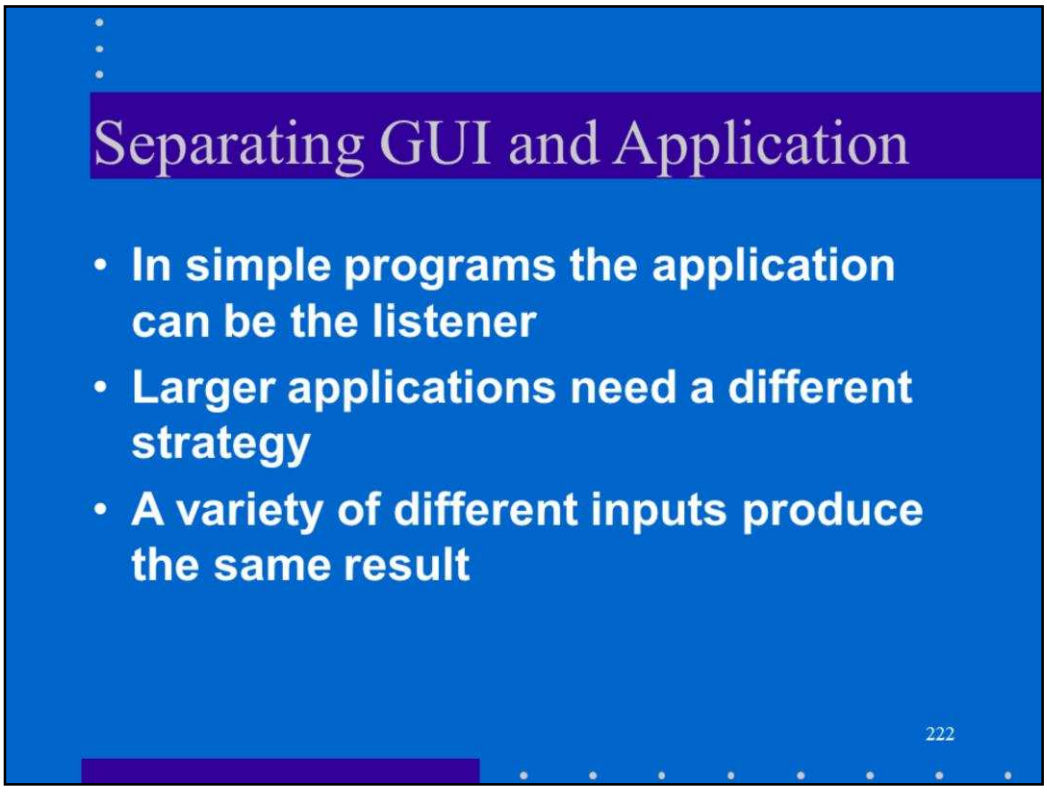
221

This version of the code uses an adapter class to cut down on the lines of code. Class `WindowCloser` extends `WindowAdapter` and contains the single method which overrides a do nothing method in `WindowAdapter`. This is the method which closes down the application.

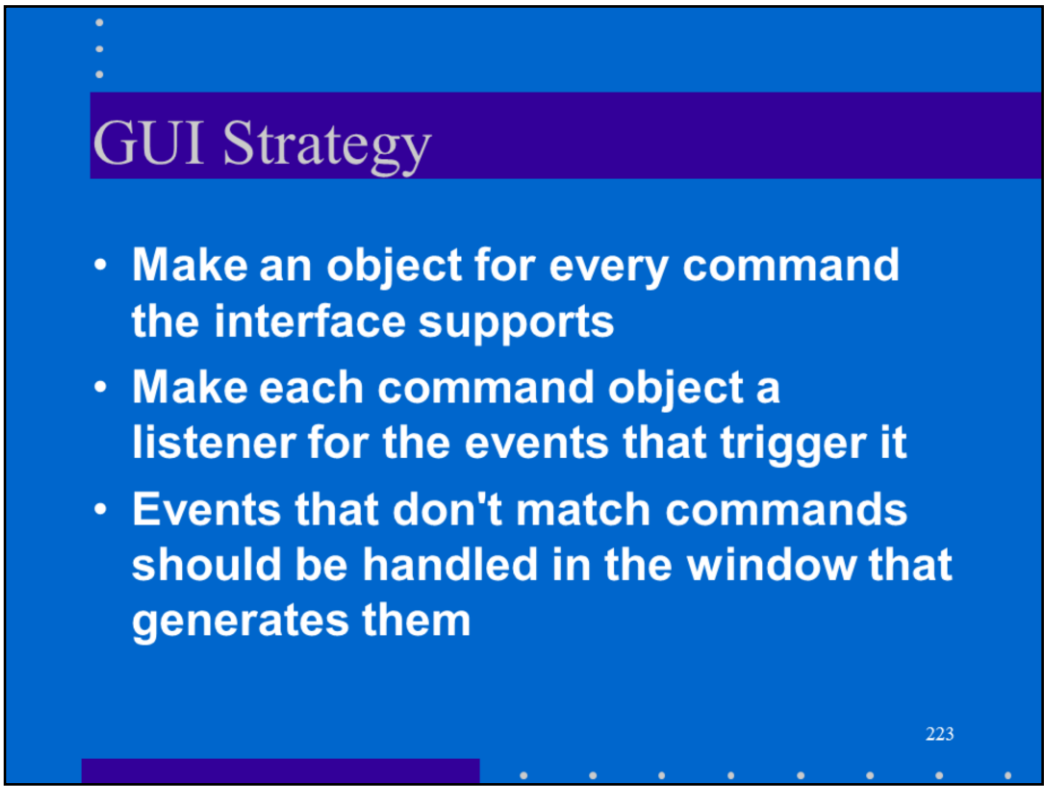
The constructor of `SecondCloseableFrame` creates an instance of `WindowCloser` to be the listener for window events rather than nominating itself as the listener.

This approach has some disadvantages in that the `WindowCloser` class cannot access any of the private instance variables of `SecondCloseableFrame`. It is very likely that this may be required. This restriction can be overcome by using anonymous inner classes which are dealt with in the next lecture.

It is worth noting that whilst it is easy to get the Java AWT classes to create a window on the screen, unless a window listener has been created then nothing other than killing off the Java program will get the window off the screen. You can press the close window button as often as you like and nothing will happen!

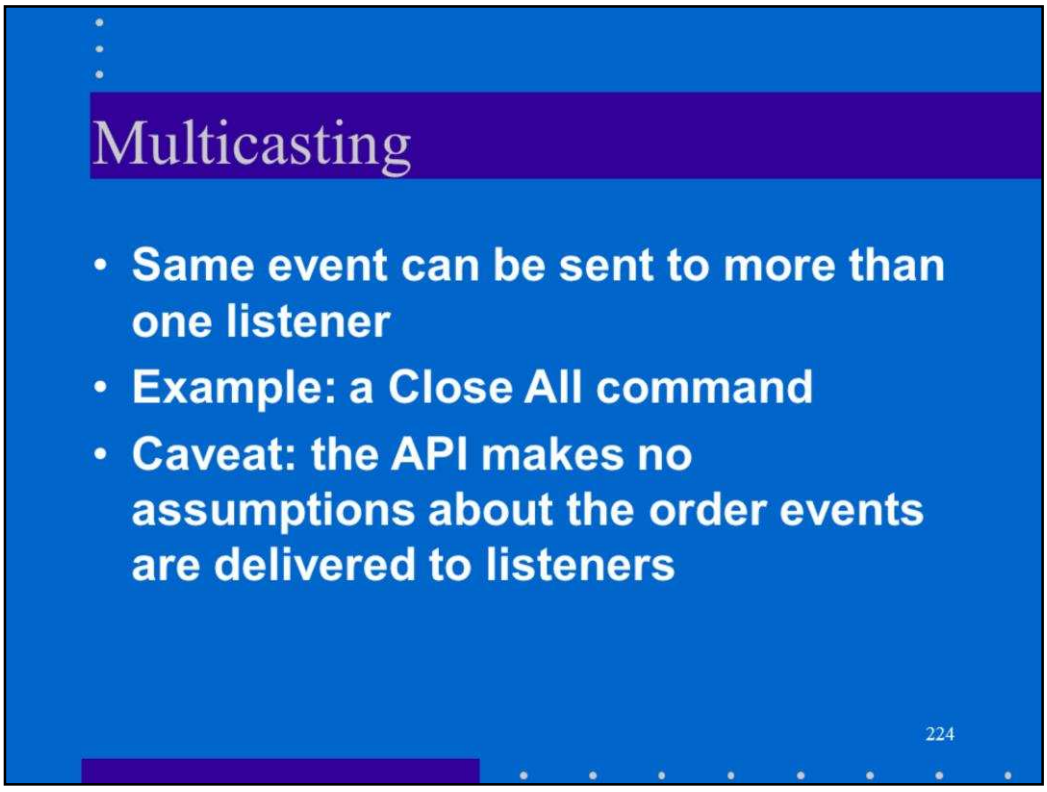

Separating GUI and Application
• In simple programs the application can be the listener
• Larger applications need a different strategy
• A variety of different inputs produce the same result
222

In most of the examples shown, the listener for events is the application. This is because the examples are small. Larger applications need to separate the interface from the application. Typically in large applications the interface must allow the user a number of different actions which will produce the same result. For example, in most Microsoft products commands can be executed via a button push, menu selection and a hot key combination. It makes sense to code this once and direct any events which require the same action to this listener.

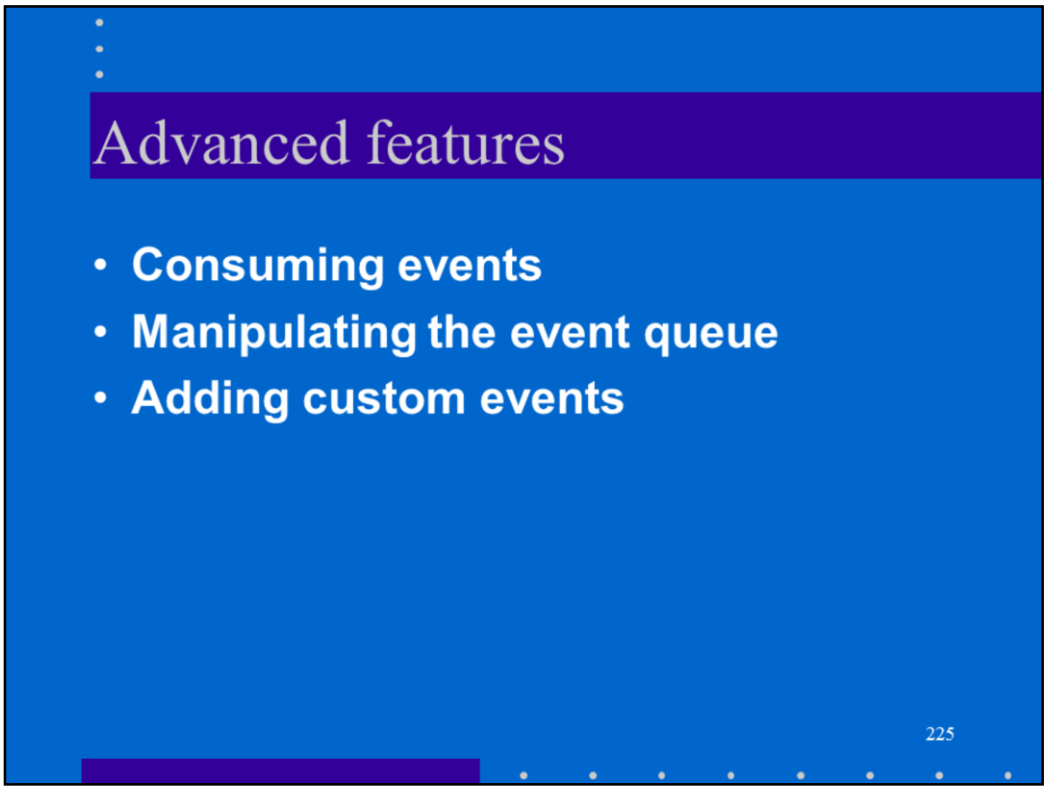

GUI Strategy
• Make an object for every command the interface supports
• Make each command object a listener for the events that trigger it
• Events that don't match commands should be handled in the window that generates them
223

A good design tactic is to define an object for every command a user can issue. This object will listen for all the different events that can signal the execution of this command. Consequently, all the code for a command is located in the one class.

In contrast events that don't relate to a command should be handled by the window in which they might occur.


Multicasting
• Same event can be sent to more than one listener
• Example: a Close All command
• Caveat: the API makes no assumptions about the order events are delivered to listeners
224

An event can be handled by multiple listeners. For example, a multiple window application might allow a Close All command as an option on a menu which can be found on every window. Selecting such an option would close all open windows. This can be achieved by allowing each window to handle the events generated by all the other windows. The technique can't guarantee the order in which handlers will receive events.



Java allows quite detailed control of events but this makes the coding harder.

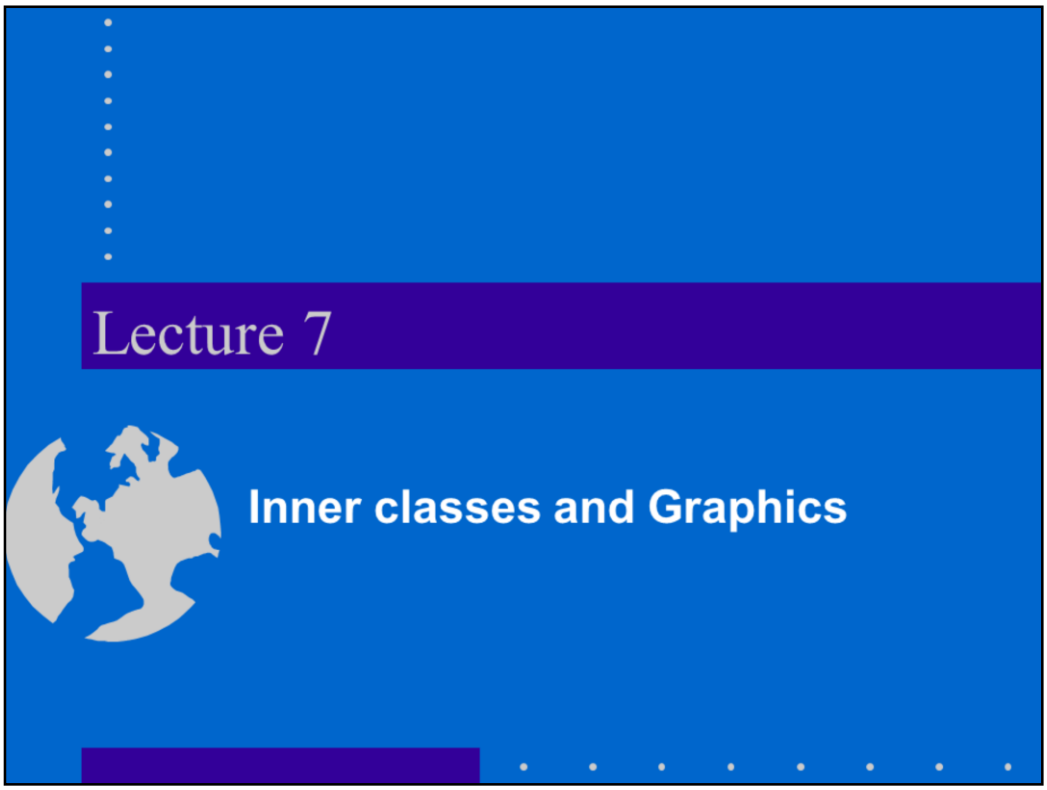
Events such as mouse clicks can be *consumed*, typically this is done by a low level event handler that wishes to stop events reaching semantic level handlers. If you were to write an application that allowed you to visually move components, you would want to suppress their normal mode of operation until they had been properly placed.

It is possible to manipulate the event queue in the same way a C/C++ program might, although this requires a great deal of knowledge about how Java optimises events in the queue.

Its possible to add custom events such as a timer event and to introduce listeners to deal with custom events.

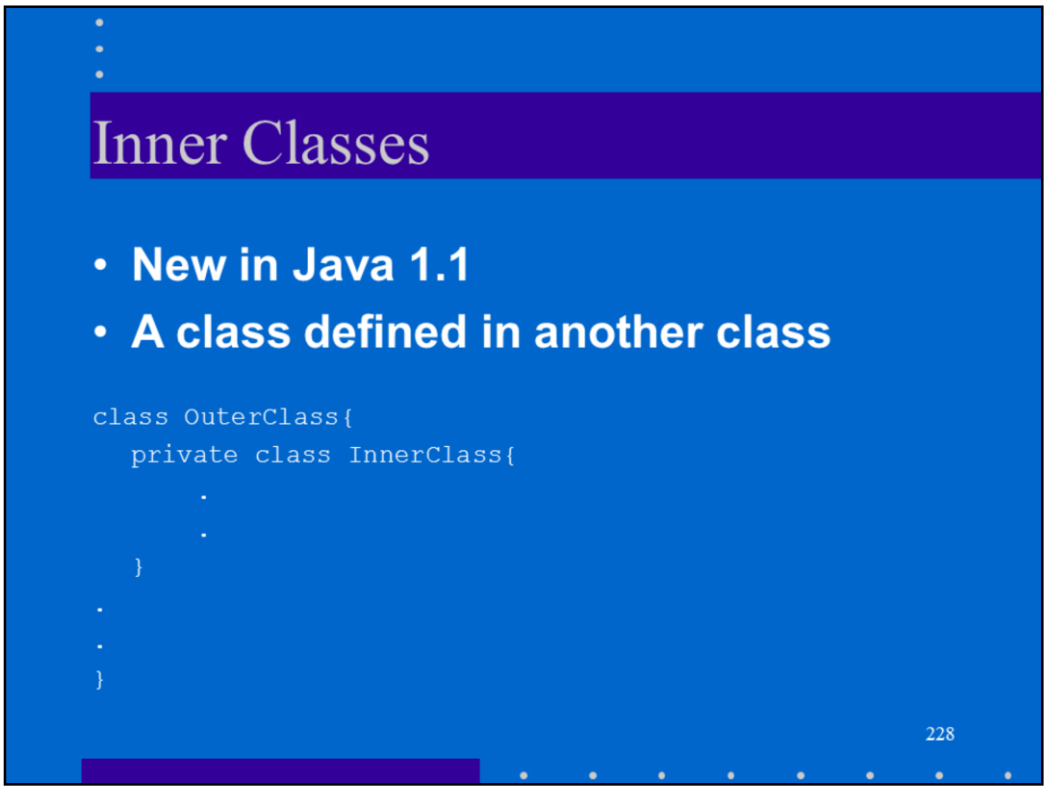
Summary

- Interfaces
- Interface variables
- Extending interfaces
- Event delegation model
- Event hierarchy
- Listeners
- Adapters
- Advanced event handling



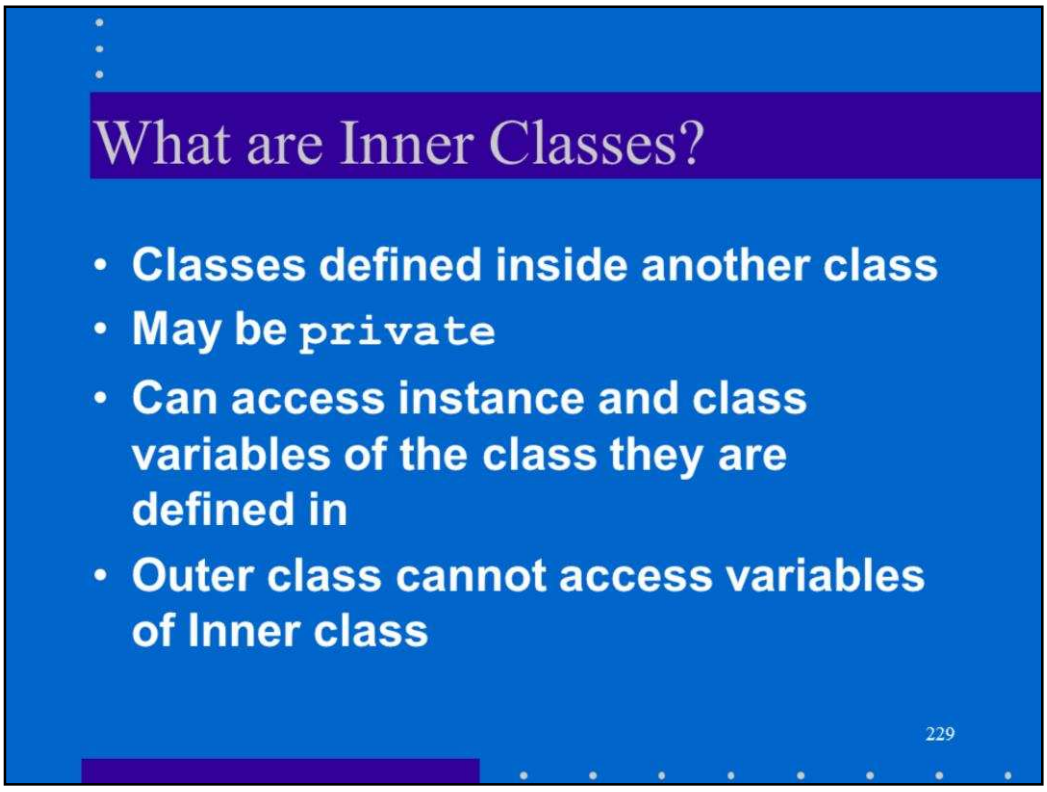
Inner classes were a new addition to Java syntax in JDK 1.1. They make it easier to write event handlers.

The rest of the lecture is an introduction to some of the Java graphics capabilities.



Inner classes were not part of the original specification for Java. They were introduced in Java 1.1. Their existence, however, can shorten the length of code.

Essentially an inner class is a class whose definition is nested inside a standard class definition.

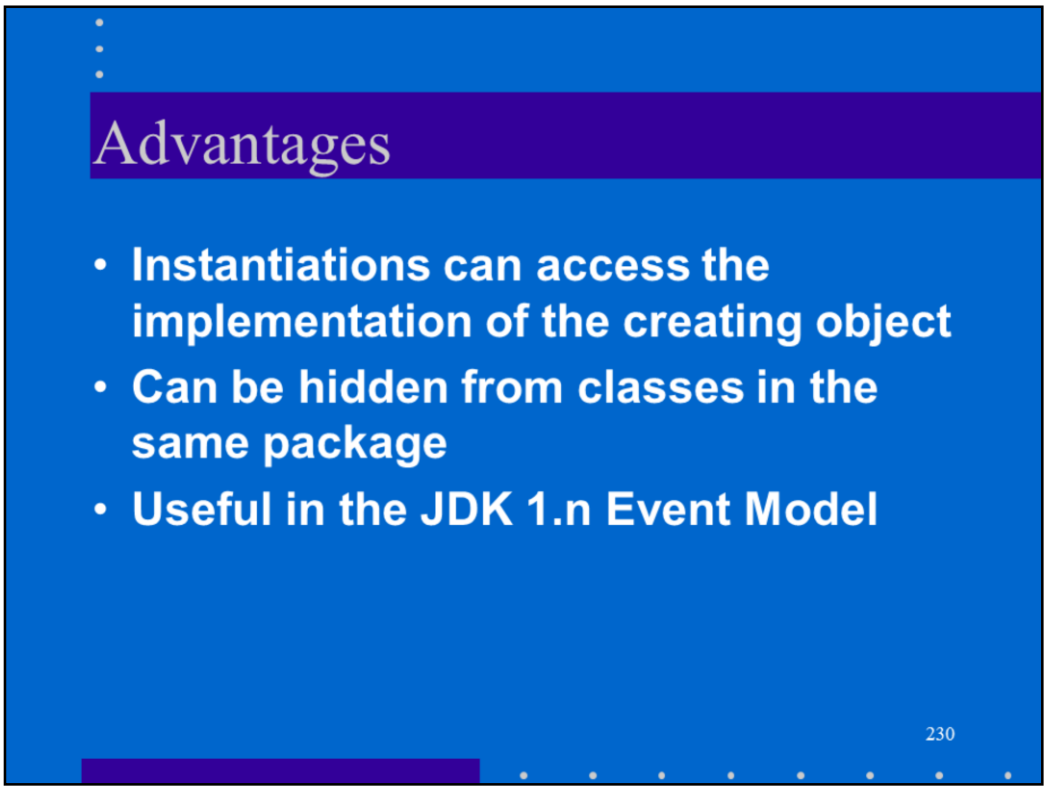

What are Inner Classes?
• Classes defined inside another class
• May be private
• Can access instance and class variables of the class they are defined in
• Outer class cannot access variables of Inner class
229

An inner class is simply a class whose definition appears inside the definition of another class. Such a definition may appear inside or outside a method definition.

Inner classes may be declared as private, other classes may not.

The scope of an inner class includes the methods and data of the class it is defined in.

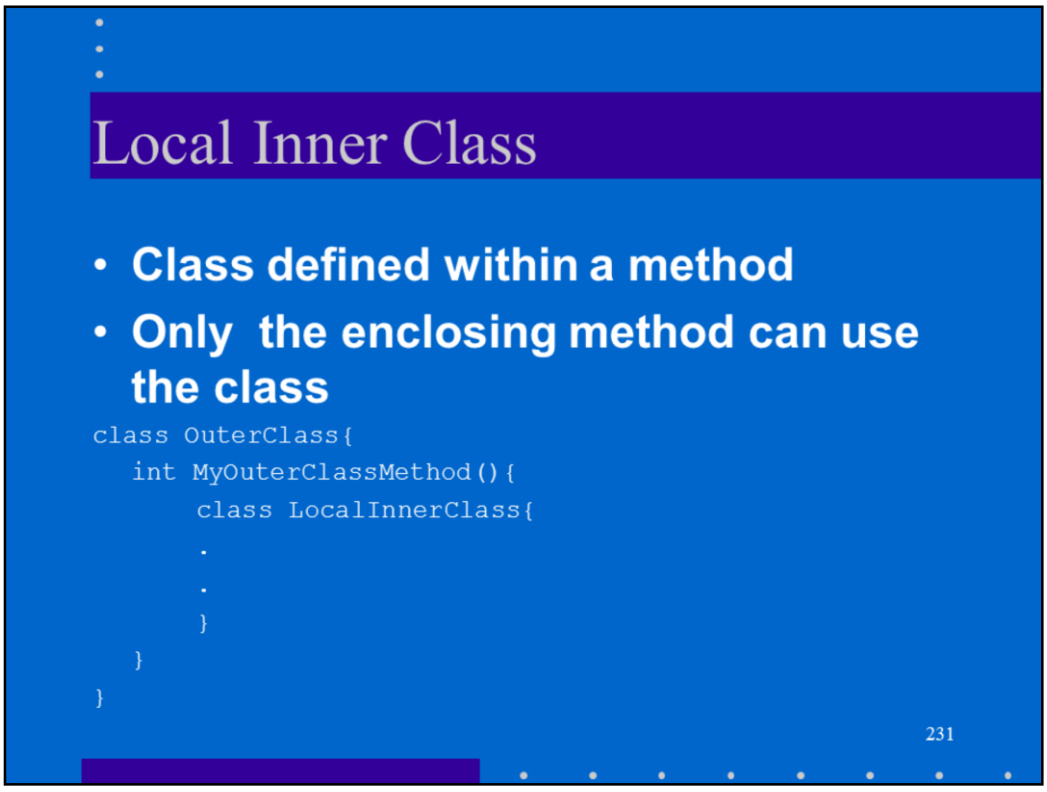
A class may not reference variables or methods of an inner class that is defined within it.


Advantages
• Instantiations can access the implementation of the creating object
• Can be hidden from classes in the same package
• Useful in the JDK 1.n Event Model
230

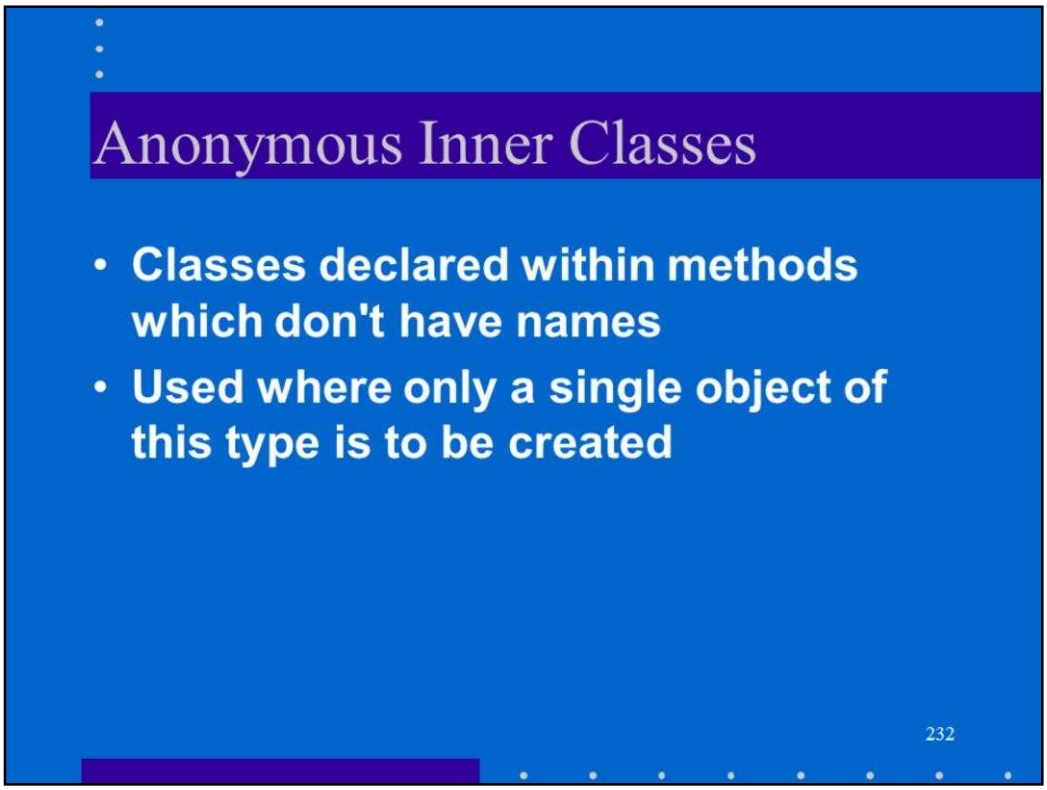
An inner class is identical to any other class in that objects can be instantiated from it. The objects created by instantiating an inner class may reference data and methods defined in the class in which the inner class is nested, even if those methods and data are `private`.

A inner class can be declared as `private` which means that other classes in the same package cannot make use of it. Objects of such classes can only be instantiated by objects from the classes they are nested in. Normal classes (outer classes) may not be `private`.

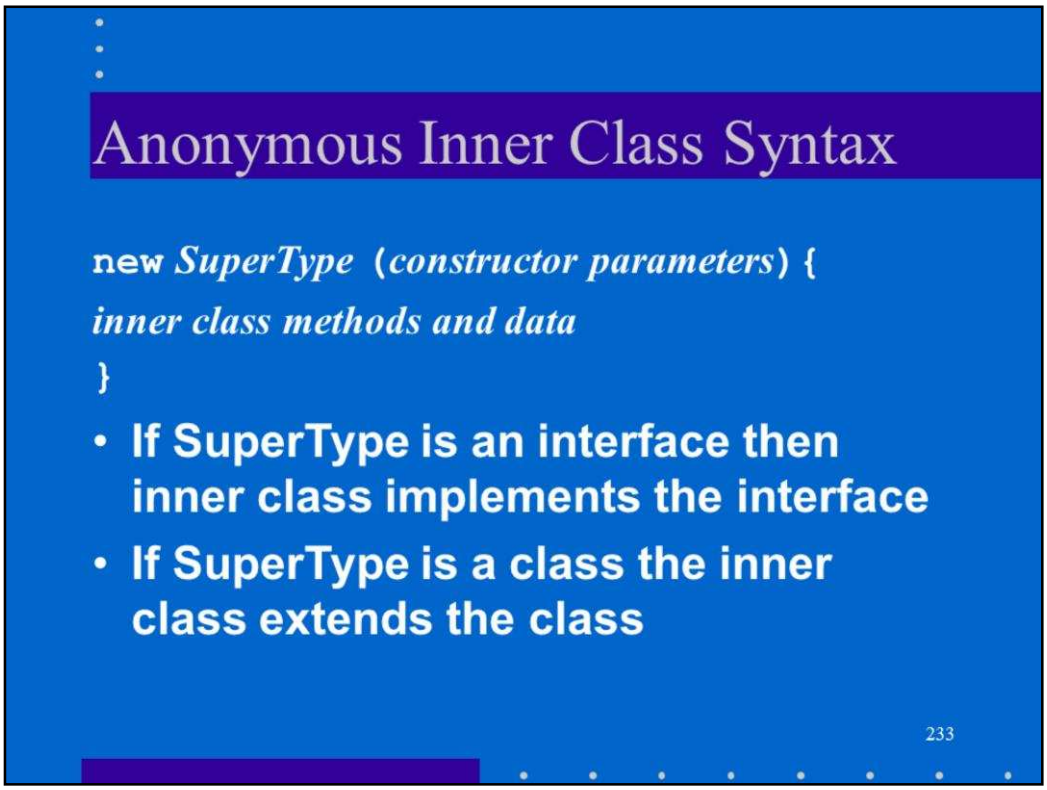
Anonymous inner classes can be used to shorten the code necessary to make use of the JDK 1.1 event model.



A class defined within a method declaration is called a *local inner class.* Objects of a local inner class may only be instantiated and used within the method in which the class definition appears.


Anonymous Inner Classes
• Classes declared within methods which don't have names
• Used where only a single object of this type is to be created
232

A local inner class often does not require a name because only one object of this class will be created within the method. Such classes are called *anonymous inner classes*. An anonymous inner class must always be associated with a statement containing the `new` operator.


Anonymous Inner Class Syntax
new SuperType (constructor parameters) {
inner class methods and data
}
• If SuperType is an interface then inner class implements the interface
• If SuperType is a class the inner class extends the class
233

Anonymous inner classes are defined by code in a block following the use of the `new` operator and an associated constructor. As we have seen, `new` may be used with the name of a class or the name of an interface.

If an anonymous inner class is defined in a block following the application of new on class X then it defines a class that extends class X.

If an anonymous innerclass is defined in a block following the application of new to interface Y the it defines a class which implements interface Y.

If X is a class then:

```
new X {.....}
```

is equivalent to creating an object of a class which extends X. The code may add to the methods of X or override them.

If Y is an interface then

```
new Y{.....}
```

is equivalent to creating an object of a class that implements Y. In this case the code defined in the block will contain definitions of the methods specified by interface Y.

Event Handling

```
import java.awt.*;
import java.awt.event.*;
class WindowCloser extends WindowAdapter{
   public void windowClosing(WindowEvent e){
        System.exit(0);}
}
public class SecondCloseableFrame extends Frame{
   public SecondCloseableFrame(){
        setSize(300, 200);
        addWindowListener(new WindowCloser());
   }
   public static void main(String[] args){
        Frame f = new SecondCloseableFrame();
        f.show();
   }
}
```

234

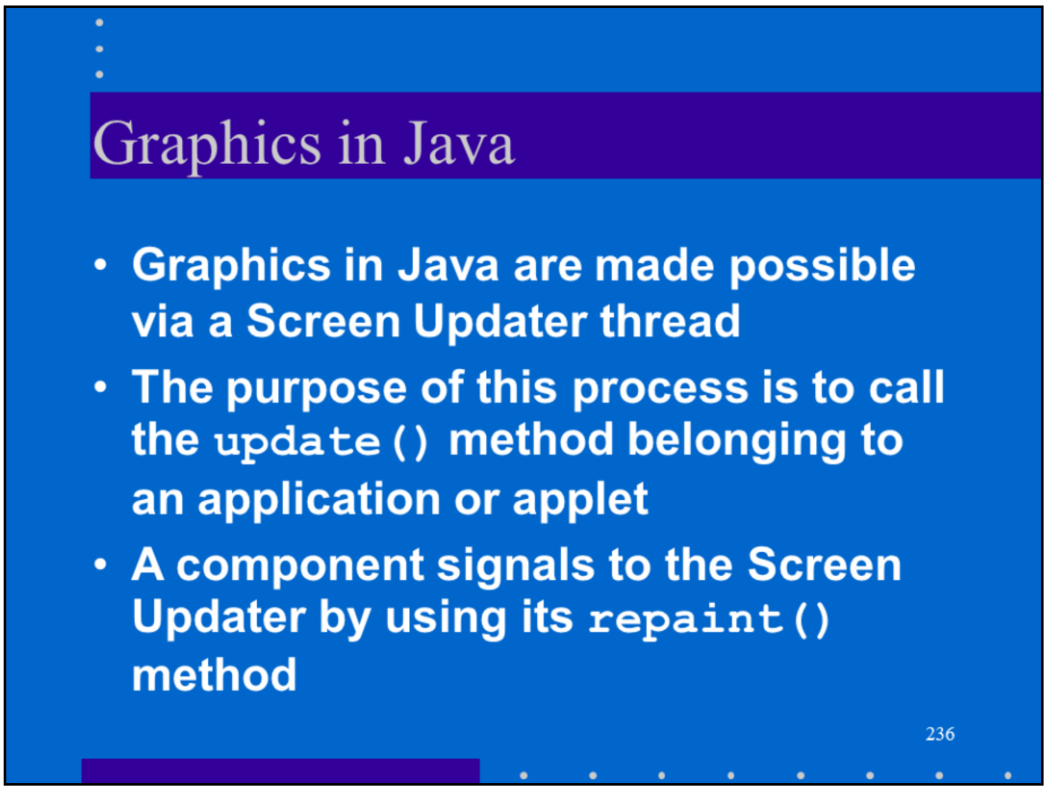
This is the code presented in the last lecture for dealing with the window closing event.

Event Handler with Inner Class

```
import java.awt.*;
import java.awt.event.*;
public class CloseableFrame extends Frame{
  public CloseableFrame(){
       addWindowListener(new WindowAdapter(){
              public void windowClosing(WindowEvent e){
                     System.exit(0);
              }
       }
       );
       setSize(300, 200);
   }
   public static void main(String[] args){
       Frame f = new CloseableFrame();
       f.show();
   }
}
```

235

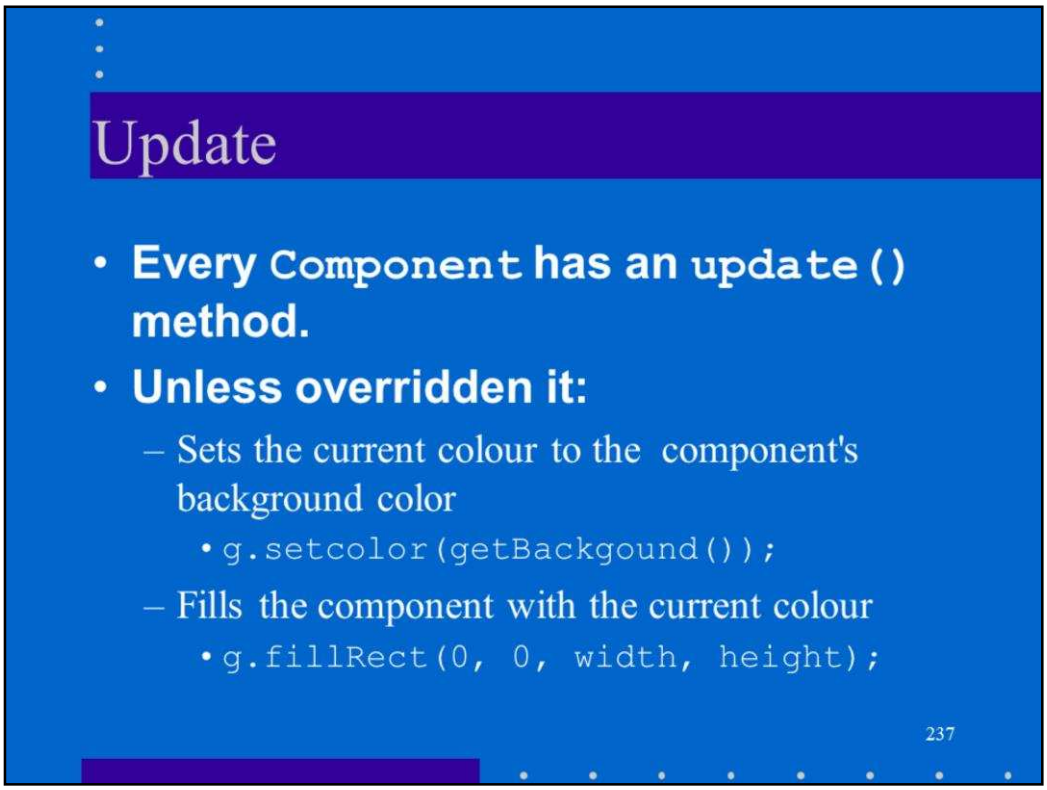
This code here performs the same task with one less class. The class `WindowCloser` has become an anonymous inner class in this example. In the previous version `WindowCloser` extended `WindowAdapter` by overriding one method. In this code this overriding takes place in the anonymous inner class.


Graphics in Java
• Graphics in Java are made possible via a Screen Updater thread
• The purpose of this process is to call the update() method belonging to an application or applet
• A component signals to the Screen Updater by using its repaint() method
236

There is a process in the Java Virtual Machine called Screen Updater which is responsible for refreshing the graphical output from Java applets and applications.

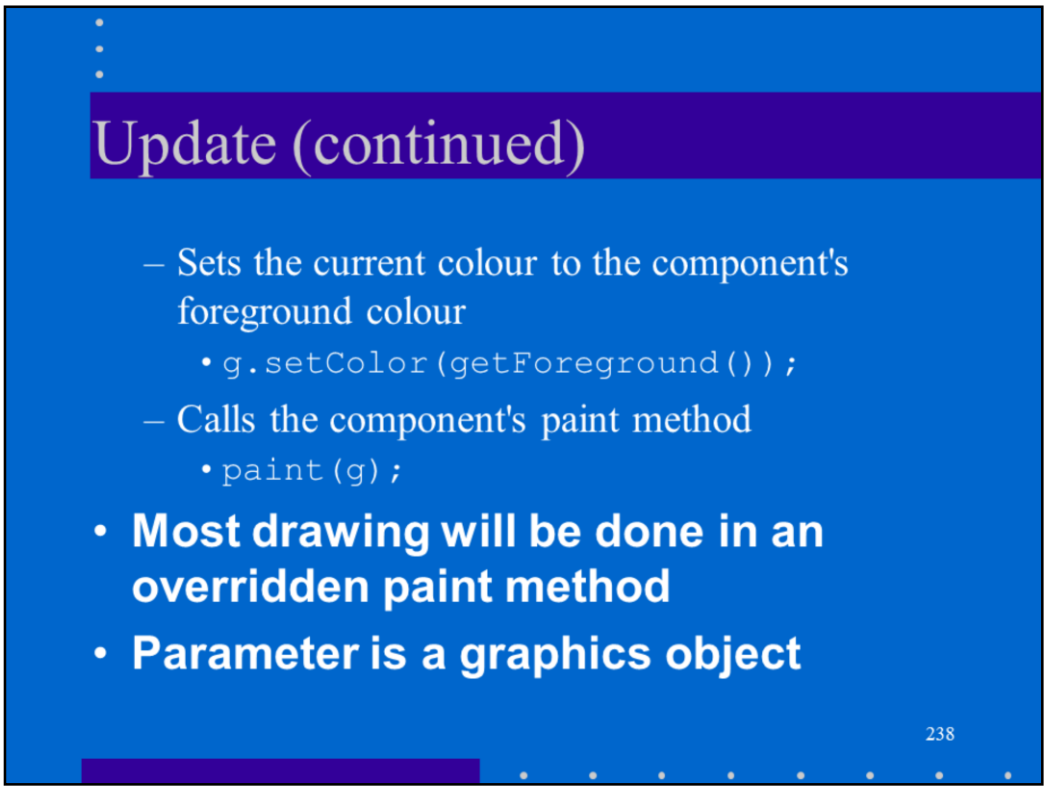
Applets that produce graphical output will override the `paint()` and/or `update()` methods for the components they are displaying. Screen Updater is responsible for calling `update()` at appropriate times.

Components can ask Screen Updater for a redraw by invoking their `repaint()` method which will add them to Screen Updaters list of tasks.


Update
• Every Component has an update() method.
• Unless overridden it:
– Sets the current colour to the component's background color
• g.setcolor(getBackgound());
– Fills the component with the current colour
• g.fillRect(0, 0, width, height);
237

Components will either inherit an `update()` method or else override it. Periodically Screen Updater will invoke `update()`.

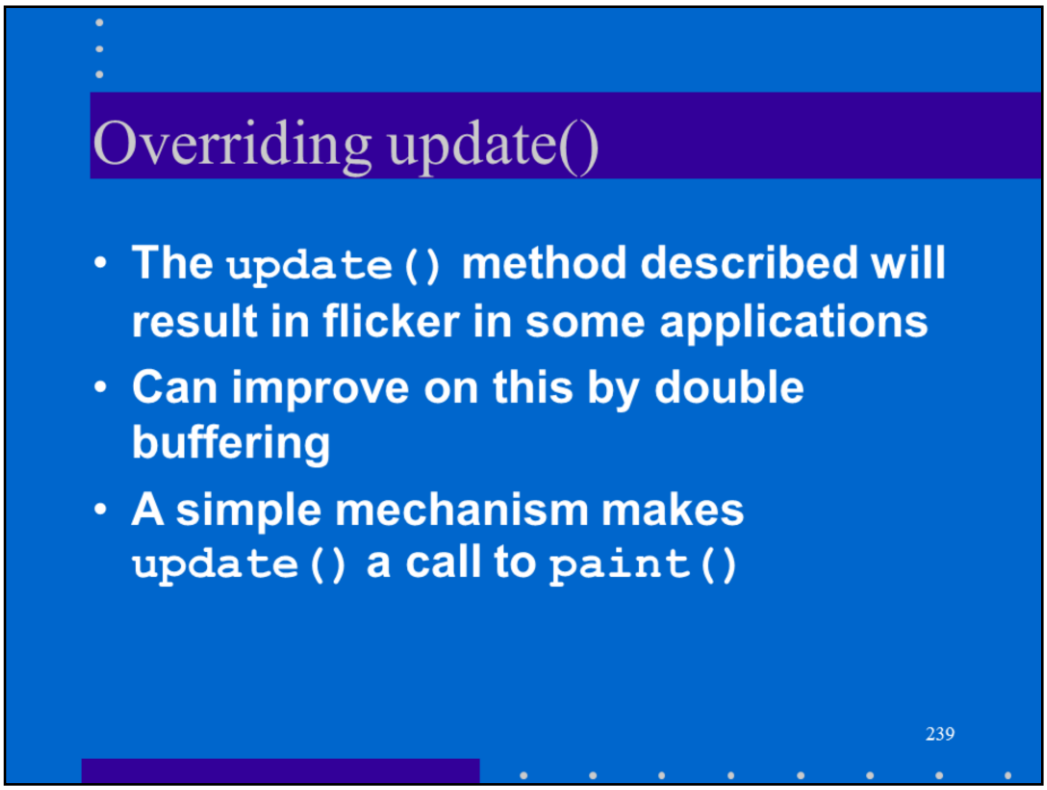
The inherited `update()` method first of all erases any graphics in the component by drawing over it in the background colour. In the case of an animation this ensures that a picture does not leave a 'trail' behind as it moves.


Update (continued)
– Sets the current colour to the component's foreground colour
• g.setColor(getForeground());
– Calls the component's paint method
• paint(g);
• Most drawing will be done in an overridden paint method
• Parameter is a graphics object
238

Having erased what was there it then invokes `paint()` to draw a new image in the standard foreground colour.

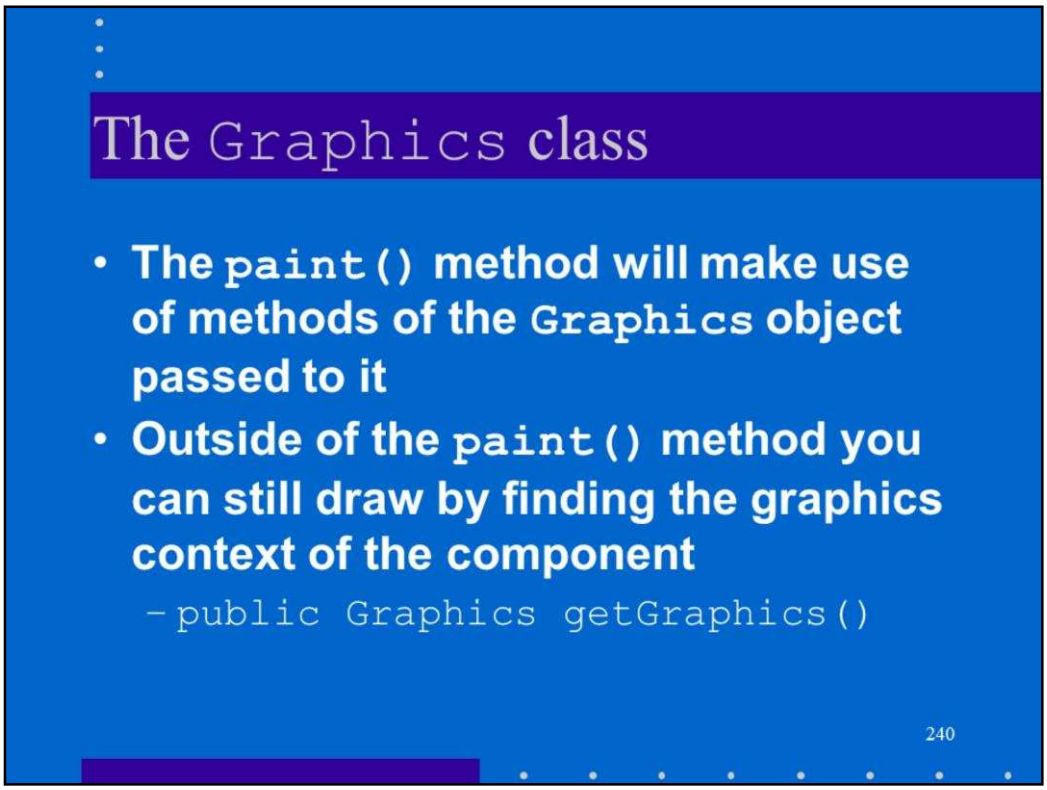
It follows then, that most of the code to draw in an application or applet will be put into an overridden version of `paint()`. The results of any drawing code outside `paint()` will normally be erased when `update()` is invoked.

`update()` passes a graphics object to `paint()`. This is the object that represents the graphical aspects of the window component. Messages to this object produce graphical output.


Overriding update()
• The update() method described will result in flicker in some applications
• Can improve on this by double buffering
• A simple mechanism makes update() a call to paint()
239

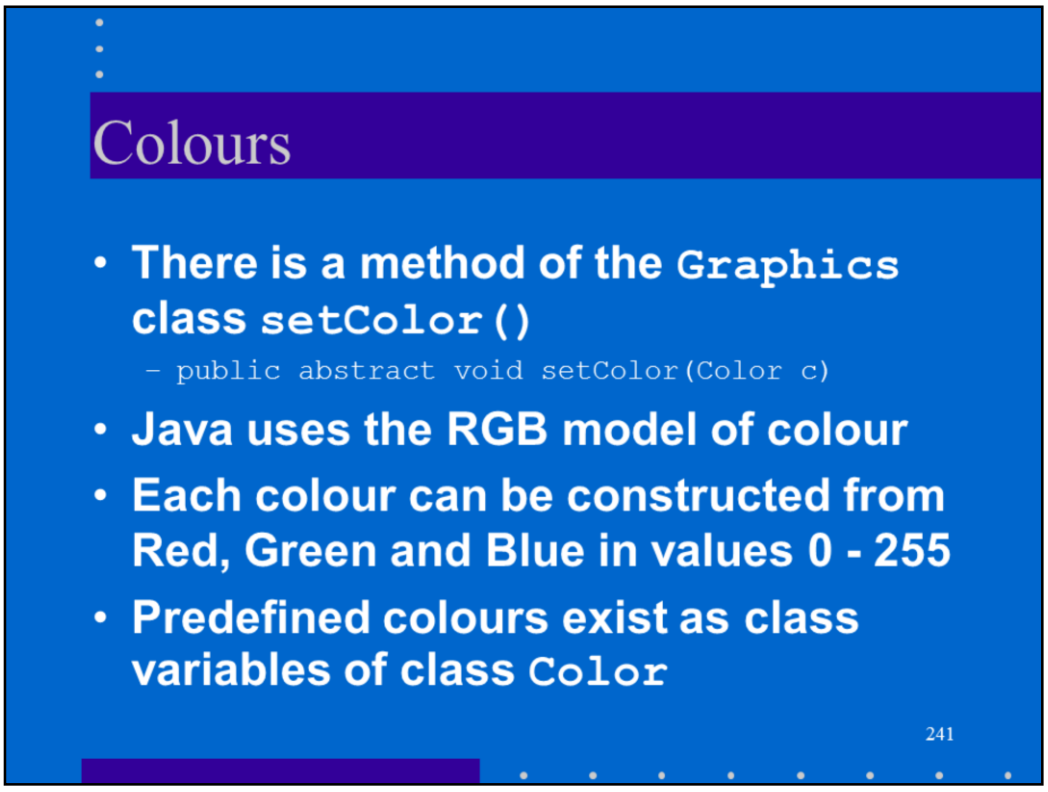
In some applications erase followed by a redraw will result in flicker. Typically, this is the case in complicated applications. Flicker can be reduced by buffering graphical output.

A simple technique is simply to override `update()` so that it doesn't erase prior to calling `paint()`. This requires caution, however, as it can lead to images lasting longer than they should.


The Graphics class
• The paint() method will make use of methods of the Graphics object passed to it
• Outside of the paint() method you can still draw by finding the graphics context of the component
– public Graphics getGraphics()
240

`paint()` produces graphical output by applying appropriate method invocations to the object of class `Graphics` which is passed to it. This object is in effect the graphic context of the screen component.

If the message `getGraphics()` is sent to any AWT component a `Graphics` object which represents the graphics context of the component is returned. Using this technique it is possible to manipulate graphics outside of a `paint()` method. It is important to realise, however, that only `update()` is called by the Screen Updater. If changes to the graphics context are to be performed outside of `paint()`, then it is likely that `update()` will require to be overridden.


Colours
There is a method of the Graphics class setColor()
– public abstract void setColor(Color c)
Java uses the RGB model of colour
Each colour can be constructed from Red, Green and Blue in values 0 - 255
Predefined colours exist as class variables of class Color
241

Colour is selected via the method `setColor()`. There are a number of specialised graphics contexts, each one of these may implement the abstract method `setColor()` in a different way.

A colour can be chosen using the RGB model of colour. In this model an intensity is selected for each of the three colour guns (Red, Green and Blue) of the monitor or printer. Intensity values range from 0 (colour not in use) to 255.

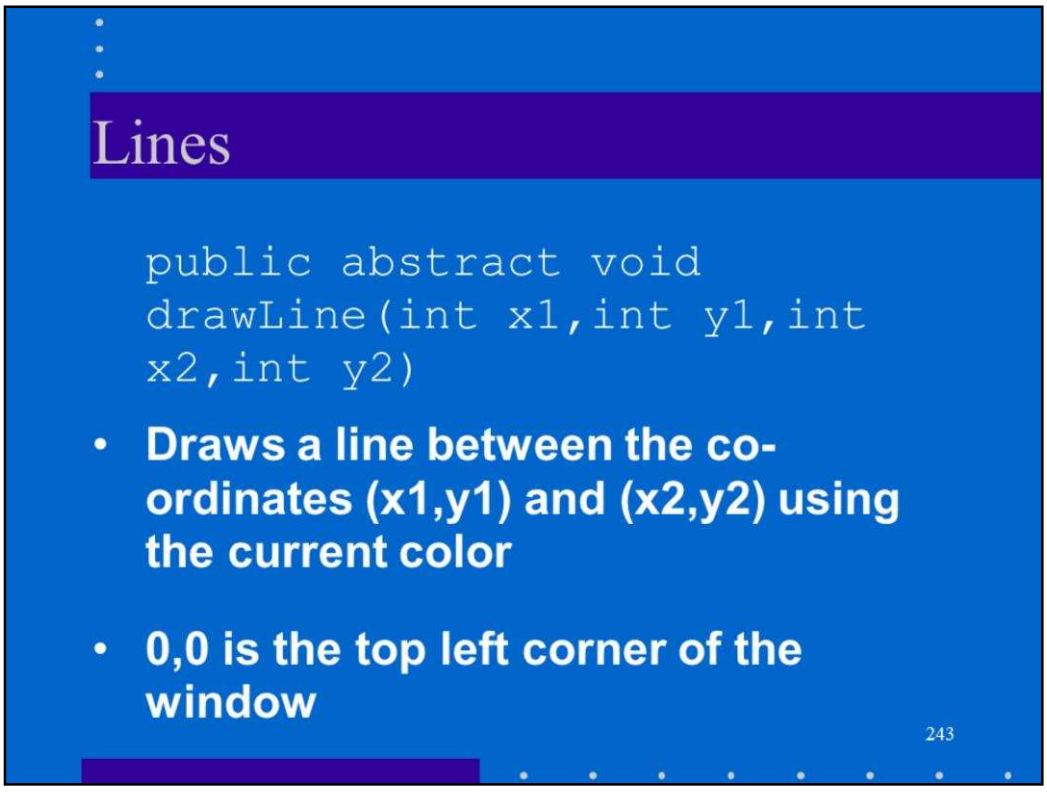
Predefined colours exist as static variables of the class `Color`.

The results of any drawing method following an invocation of `setColor()` will be in the colour chosen.

`setColor()` accepts a parameter which is an object of class `Color`.

Class `Color` has three constructors each of which creates an object representing a colour. The constructors accept parameters which specify RGB values.

This slide list the class variables that represent predefined colours in class `Color`. Their associated RGB values are shown as well.


Lines
public abstract void drawLine(int x1,int y1,int x2,int y2)
• Draws a line between the co-ordinates (x1,y1) and (x2,y2) using the current color
• 0,0 is the top left corner of the window
243

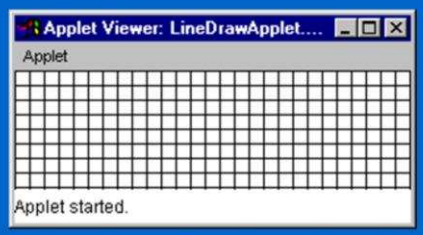
`drawline()` is a method of class `Graphics` which can be used to draw a line between two co-ordinates. Lines are one pixel thick. The Java documentation says:
Co-ordinates are infinitely thin and lie between the pixels of the output device. Operations which draw the outline of a figure operate by traversing along the infinitely thin path with a pixel-sized pen that hangs down and to the right of the anchor point on the path. Operations which fill a figure operate by filling the interior of the infinitely thin path. Operations which render horizontal text render the ascending portion of the characters entirely above the baseline co-ordinate.

Some important points to consider are that drawing a figure that covers a given rectangle will occupy one extra row of pixels on the right and bottom edges compared to filling a figure that is bounded by that same rectangle. Also, drawing a horizontal line along the same y co-ordinate as the baseline of a line of text will draw the line entirely below the text except for any descenders. Both of these properties are due to the pen hanging down and to the right from the path that it traverses.

All co-ordinates which appear as arguments to the methods of this Graphics object are considered relative to the translation origin of this Graphics object prior to the invocation of the method. All rendering operations modify only pixels which lie within the area bounded by both the current clip of the graphics context and the extents of the Component used to create the Graphics object.

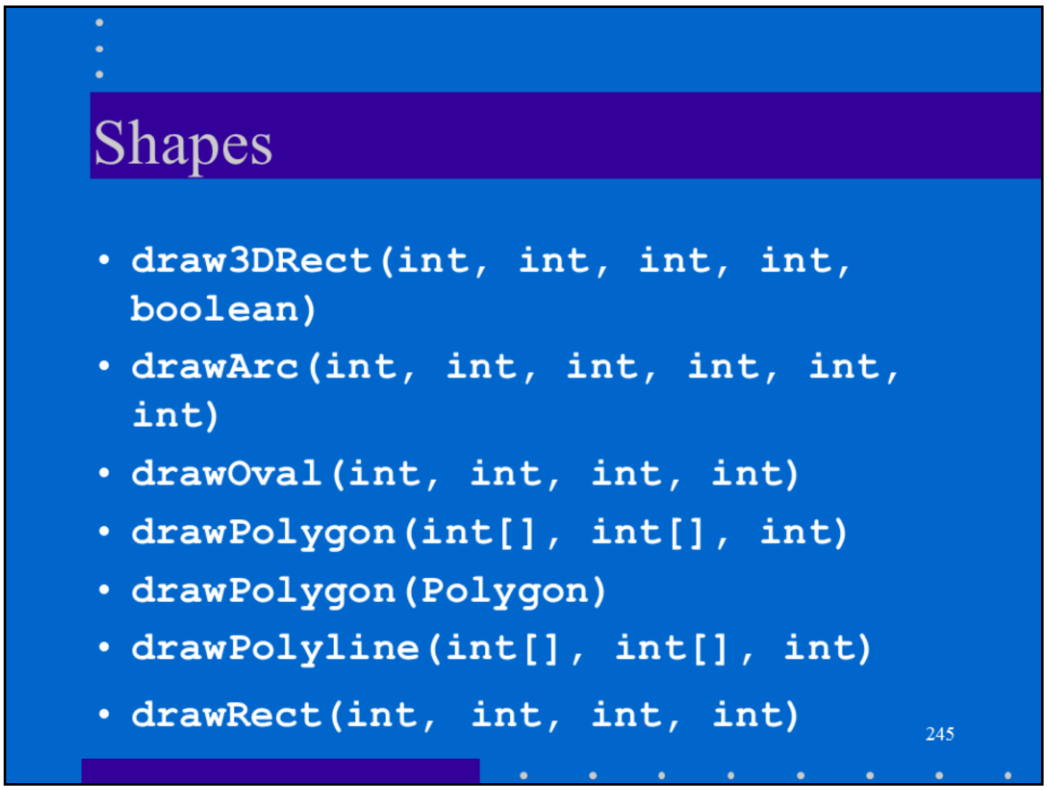
Line Example

```
import java.awt.*;
import java.applet.*;
public class LineDrawApplet extends Applet{
   public void paint(Graphics g){
        Rectangle r=getBounds();
        for(int x=0;x<r.width;x=x+10)
              g.drawLine(x,0,x,r.height);
        for(int y=0;y<r.height;y=y+10)
              g.drawLine(0,y,r.width,y);
   }
}
```



This applet overrides the `paint()` method with code that draws equally spaced lines across the window.

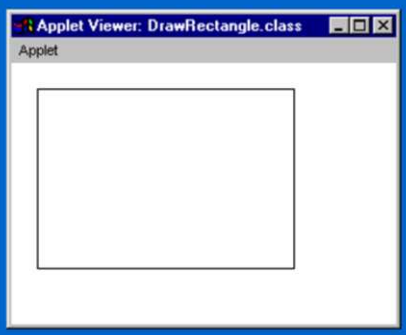
The `getBounds()` method returns a `Rectangle` which is the same size as the window. A `Rectangle` responds to the methods `width()` and `height()` returning its width and height in pixels respectively.



These are some of the methods for drawing the outlines of shapes. Similar methods exist for filled shapes. Full details are in the API documentation.

Shape Example

```
import java.awt.*;
import java.applet.*;
public class DrawRectangle extends Applet{
   public void paint(Graphics g){
       g.drawRect(20,20,200,140);
   }
}
```

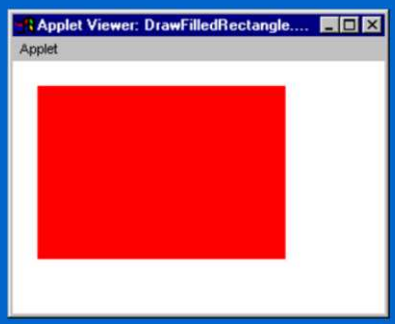


246

This applet overrides `paint()` to draw the out line of a rectangle. The rectangle has its diagonal corners at pixel 20,20 and pixel 200,140. Pixel 0,0 is the pixel at the top left hand corner of the window.

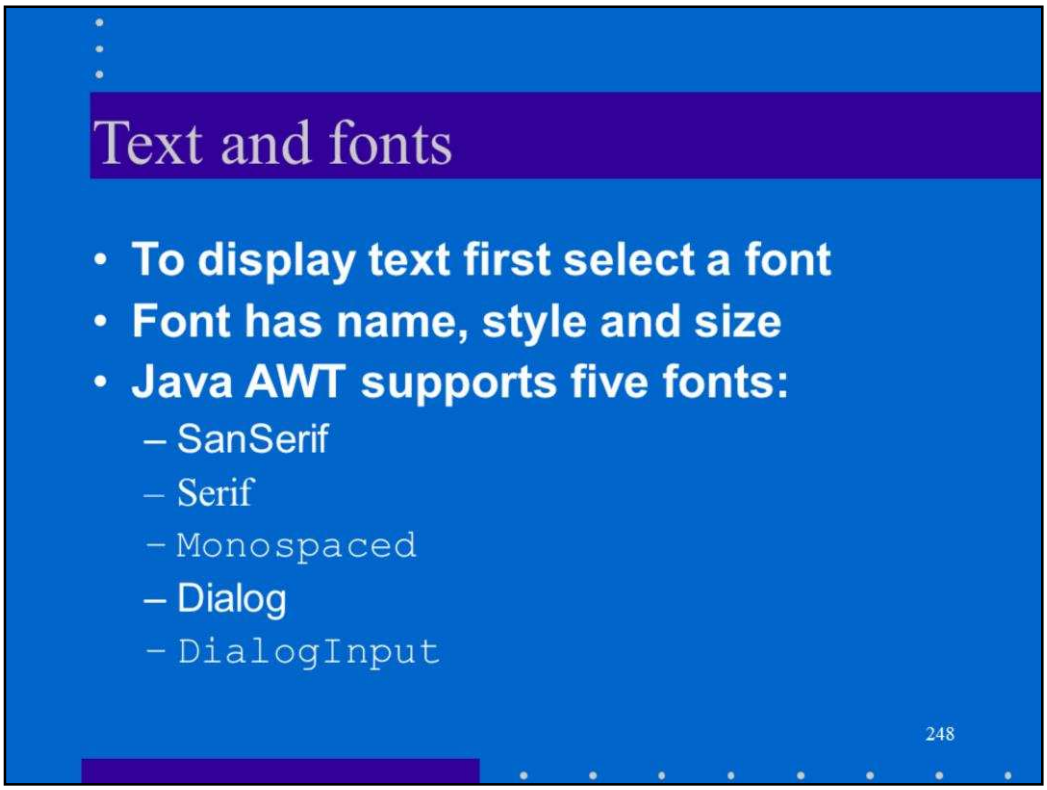
Filled Shapes

```
import java.awt.*;
import java.applet.*;
public class DrawFilledRectangle extends Applet{
   public void paint(Graphics g){
        g.setColor(Color.red);
        g.fillRect(20,20,200,140);
   }
}
```



247

This applet sets the colour and then draws a filled rectangle.



Text is associated with a font. Every font has a name, a style (plain, bold. italic or bold italic) and a size in points.

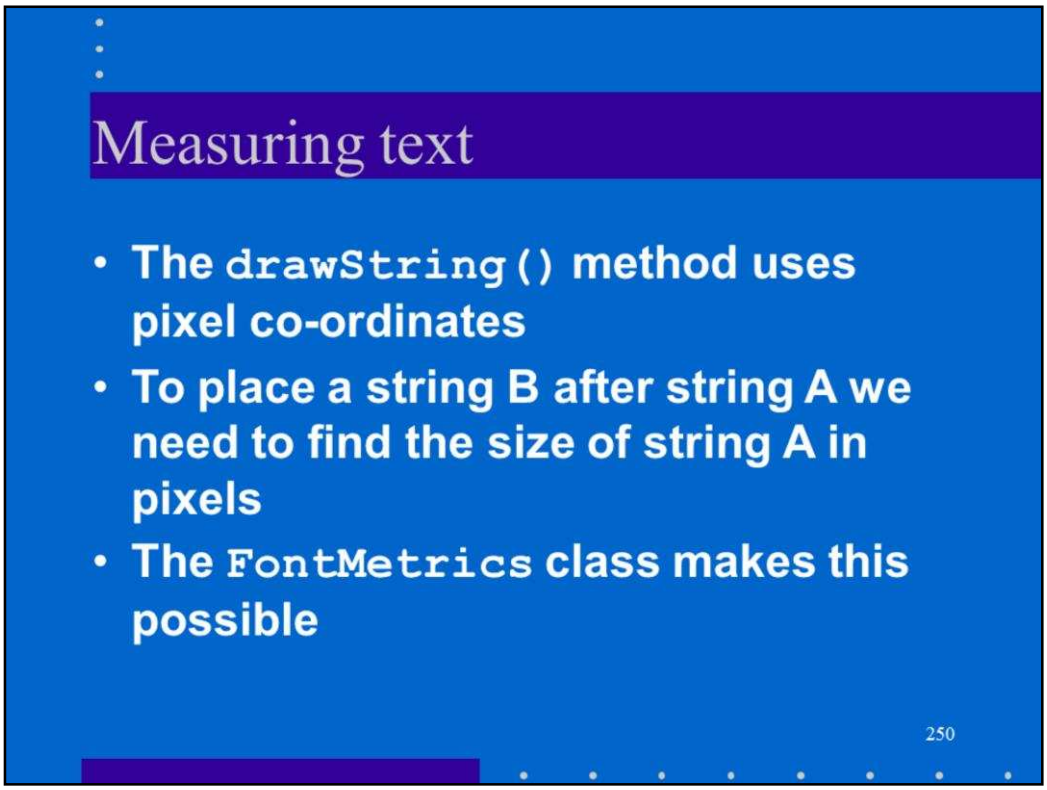
For portability there are only five standard fonts in Java. For each JVM implementation there is a default mapping to fonts supported for on that platform. For example, on Windows SanSerif is mapped to Arial. It is possible to change this default mapping and get access to many more fonts. This process is a little complicated though!

Font Example

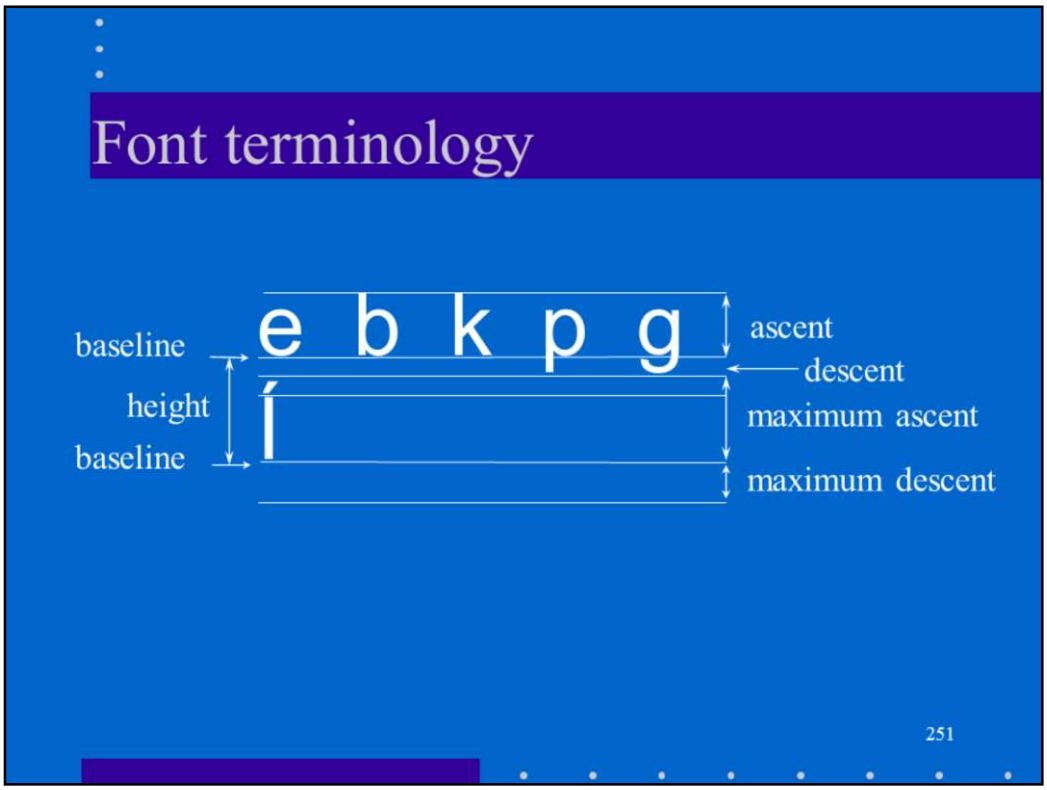
```
import java.awt.*;
import java.applet.*;
public class FontTest extends Applet{
   public void paint(Graphics g){
        Font f = new Font("SansSerif",Font.BOLD,20);
        g.setFont(f);
        g.drawString("Hello World",25,25);
   }
}
```

249

This applet creates an instance of class Font which specifies a bold, twenty point SansSerif font. It then sets this as the current font. The font is used to output the string "Hello World" at pixel 25,25.


Measuring text
• The drawString() method uses pixel co-ordinates
• To place a string B after string A we need to find the size of string A in pixels
• The FontMetrics class makes this possible
250

There is a complication which occurs if you wish to mix fonts on a screen (a bold version of the same type face is a different font). The `drawString()` method allows you to place the beginning of a text string. Where it ends will depend on the size of the characters and the space between them. Two strings with the same number of characters (but different characters) in them will almost certainly have different display lengths because the characters have different widths. For each separate font we use, we will need to use a separate invocation of `drawString()`. Placing the first string is straightforward but finding where to put the second string is not. We need to know the length of each string in pixels. The `FontMetrics` class is used for this.



Class `FontMetrics` can give measurements for many characteristics of fonts. It is based on traditional printer's measurements.

`getAscent()`
Gets the font ascent.

`getDescent()`
Gets the font descent.

`getFont()`
Gets the font.

`getHeight()`
Gets the standard height of a line of text in this font.

`getLeading()`
Gets the standard leading, or line spacing, for the font.

`getMaxAdvance()`
Gets the maximum advance width of any character in this Font.

`getMaxAscent()`
Gets the maximum ascent of all characters in this Font.

`getMaxDescent()`
Gets the maximum descent of all characters in this Font.

`stringWidth(String str)`
Returns the total advance width for showing the specified String in this Font. The advance width is the amount by which the current point is moved from one character to the next in a line of text.

.

Mixed font example(1)

```
import java.awt.*;
import java.applet.*;
public class FontMetricTest extends Applet{
   public void paint(Graphics g){
        Font f1=new Font("SansSerif",Font.BOLD,20);
        Font f2=new Font("SansSerif",
                    Font.BOLD+Font.ITALIC,14);
        FontMetrics f1m=g.getFontMetrics(f1);
        FontMetrics f2m=g.getFontMetrics(f2);
        int cx=25, cy=25;
        String s1="Hello ",s2="Italic",s3=" World";
        .
        .
        .
```

252

This applet will print "Hello " in 20 point bold SansSerif, followed by "italic" in 14 point bold italic SansSerif, followed by " World" in the original font.

The first two lines create the fonts. Note that the style is created by adding together class variables.

Two `FontMetrics` objects are created from the two fonts using the `getFontMetrics()` method in class `Graphics`. References to these objects are stored in `f1m` and `f2m`.

The co-ordinates for string positioning are held in variables `cx` and `cy`.

Mixed font example(2)

```
        .
        .
        .
        g.setFont(f1);
        g.drawString(s1,cx,cy);
        cx=f1m.stringWidth(s1)+cx;
        g.setFont(f2);
        g.drawString(s2,cx,cy);
        cx=f2m.stringWidth(s2)+cx;
        g.setFont(f1);
        g.drawString(s3,cx,cy);
    }
}
```

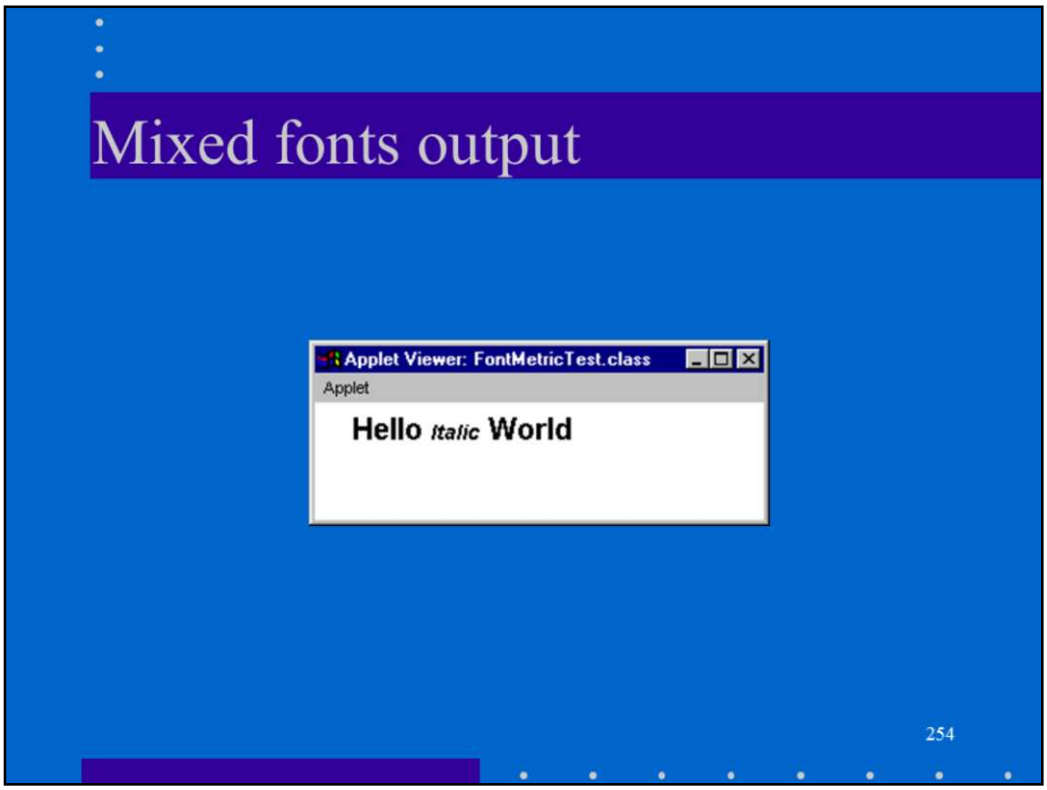
253

The first font is selected using `setFont()` and the string is drawn. The width of the first string is calculated using the `stringWidth()` method applied to the font metrics for the first font.

The x co-ordinate is incremented by the number of pixels the first string occupies.

The second font is selected. The second string is drawn and measured. The x co-ordinate is incremented by the size of the string length.

The original font is selected and the third string is drawn.



The result is text which is appropriately spaced. The middle word is displayed in a smaller italic font.

Images
• Java can display images which are stored in a file or on the Internet
• Image types supported are GIF and JPEG
• To read a file use a Toolkit object
• An Image object will hold the image
255

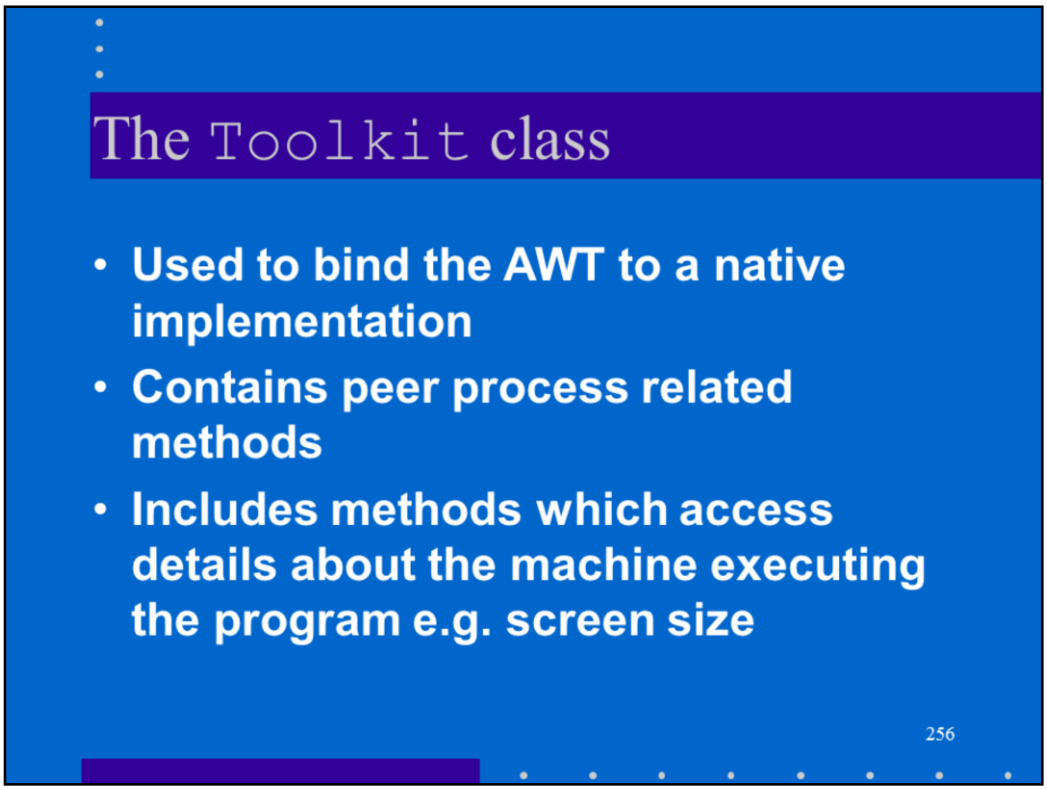
A Java applet or application can draw images which it may load from a local file or across the Internet using a URL.

Only two file formats are supported, JPEG or GIF.

An object of class `Toolkit` can be used to fetch an image from a local file. An object of class `Url` (see a later lecture) can be used to fetch images from the Internet. Applets which are loaded by browsers cannot load images from local files, as this is forbidden by the Java security model. The appletviewer will permit applets to load images from files.

Images whether read from the Internet, or from a file are represented by an object of class `Image`.

It is also possible to use the `URL` class to read images from a file.


The Toolkit class
• Used to bind the AWT to a native implementation
• Contains peer process related methods
• Includes methods which access details about the machine executing the program e.g. screen size
256

The `Toolkit` class is the link between the implementation independent AWT and the implementation specific native methods. It supports methods which can be used to find the details of the implemetation platform such as the screen size in pixels. It is also used for loading files.

Image drawing example

```
import java.awt.*;
import java.applet.*;
public class ImageTest extends Applet{
   Image picture = Toolkit.getDefaultToolkit().getImage
                                    ("picture.gif");
   public void paint(Graphics g){
       g.drawImage(picture,25,25,this);
   }
}
```

257

This applet uses the default `Toolkit` object to load a GIF file and create an `Image` object to hold it. In this example the file is located in the same directory as the applet's class file.

The image is then drawn using the `drawImage()` method with its top left corner at pixel position 25,25.

```
public abstract boolean drawImage
          (Image img,
           int x,
           int y,
           ImageObserver observer)
```

Draws as much of the specified image as is currently available at the specified co-ordinate (x, y). This method will return immediately in all cases, even if the entire image has not yet been scaled, dithered and converted for the current output device. If the current output representation is not yet complete then the method will return `false` and the indicated `ImageObserver` object will be notified as the conversion process progresses.

Parameters:

`img` - the specified image to be drawn

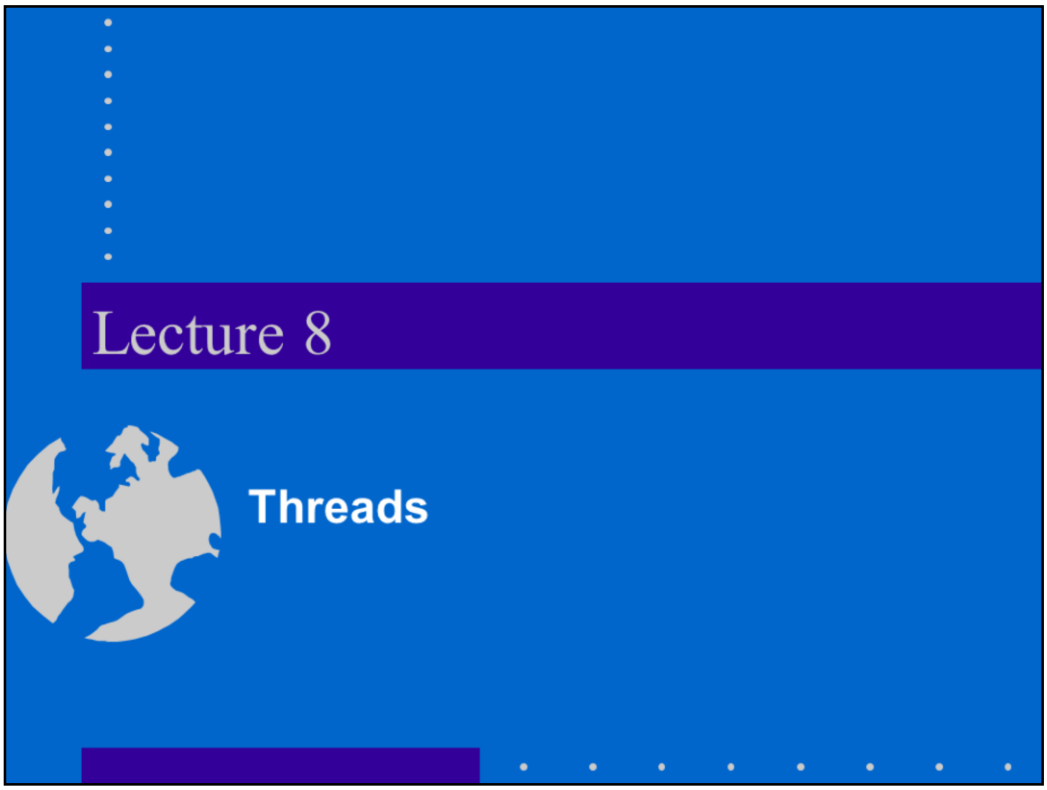
`x` - the x co-ordinate

`y` - the y co-ordinate

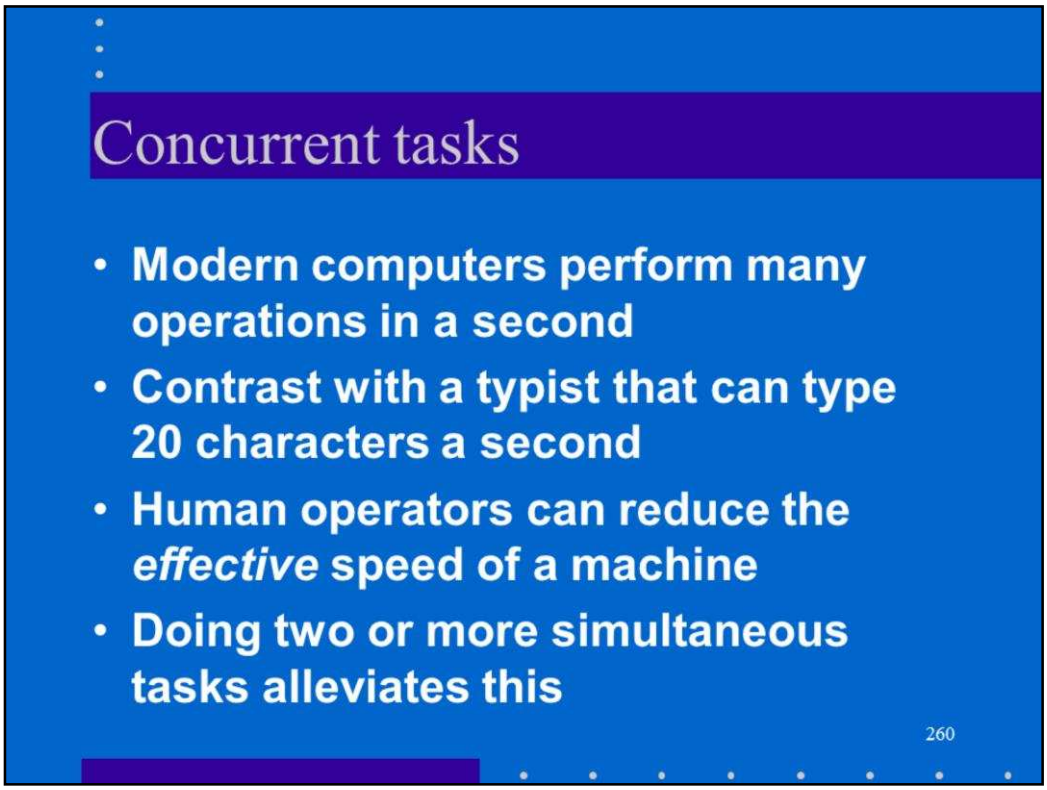
`observer` - object to be notified as more of the image is converted

Summary

- Inner classes
- Local inner classes
- Anonymous inner classes
- Graphics
- `Paint()` method
- Lines
- Shapes
- Text
- Font metrics
- Images



The topic of this lecture might once have appeared as a theoretical part of a systems programming course. Advances in modern programming languages and their associated environments have made the implementation of concurrent programming techniques much easier. Unfortunately, the same advances that make covering this material feasible have also made it almost essential as part of a programming course.

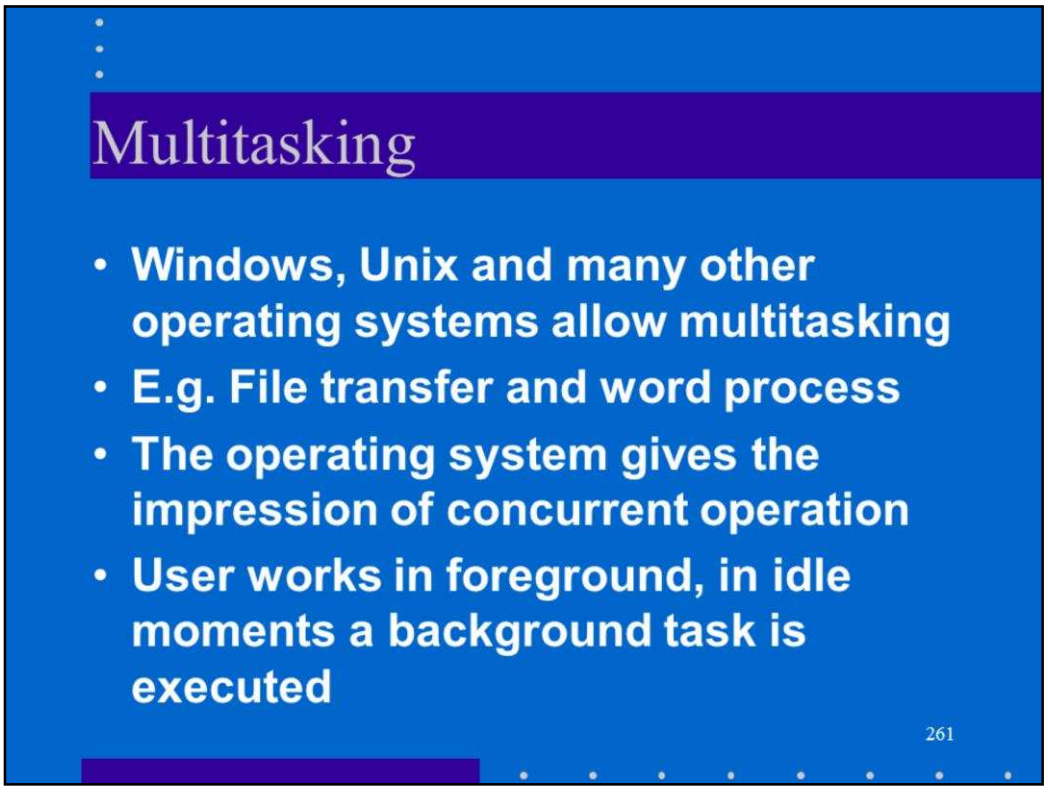

Concurrent tasks
• Modern computers perform many operations in a second
• Contrast with a typist that can type 20 characters a second
• Human operators can reduce the *effective* speed of a machine
• Doing two or more simultaneous tasks alleviates this
260

The very earliest PCs, even though in comparison with today's machines they were very slow, spent a lot of their time doing nothing. This was because they were very fast in comparison with their users. A lot of time was spent waiting for the user to do something.

Even a fast typist cannot rival the speed at which a computer can operate. If a computer is solely engaged in running a word-processing package then it will spend a lot of time simply waiting for entries from the keyboard. If the typist is spending time thinking about what to type next then the amount of waiting becomes even greater.

The fast processor speed is still needed because the user expects to see a fast response when they finally get round to issuing a command. However, if we were to work out the number of useful instructions executed in a day we might find that the effective speed of the machine was much less than the headline speed of the chip.

The way to get more work from a computer is to give it another task to do during periods when the user is inactive. It might be possible, for instance, to down load useful files from the Internet whilst the user is deciding what to type next. If this is to be made possible it is necessary to find a mechanism which allows the computer to switch between tasks whenever the task it is currently executing becomes unable to progress.

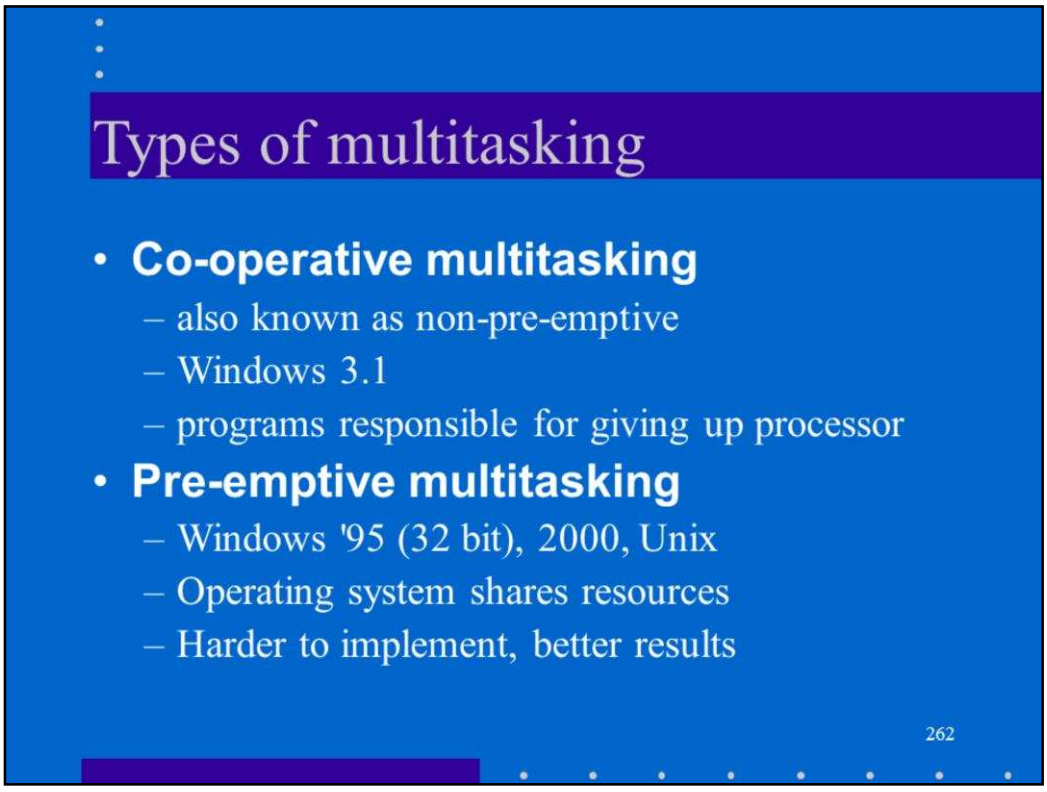

Multitasking
• Windows, Unix and many other operating systems allow multitasking
• E.g. File transfer and word process
• The operating system gives the impression of concurrent operation
• User works in foreground, in idle moments a background task is executed
261

In the days before PCs most computers supported operating systems which allowed a number of tasks to execute alongside each other. MSDos only allowed a very limited form of multitasking. In MSDos it was just possible to use a wordprocessor and print in the background at the same time, though even this was stretching the technology. As Windows moved from version to version so its capabilities for multitasking grew and in systems such as Windows '95 and Windows NT it is normal (and necessary) to be running a number of different tasks at the same time.

Note, though that the tasks do not actually run simultaneously on most machines. There is usually only one processor in a PC and this means that only one instruction can be executed at any time. The operating system gives the impression that multiple tasks are running by rapidly switching between eligible tasks. To the human eye, it appears that many tasks are performed at once.

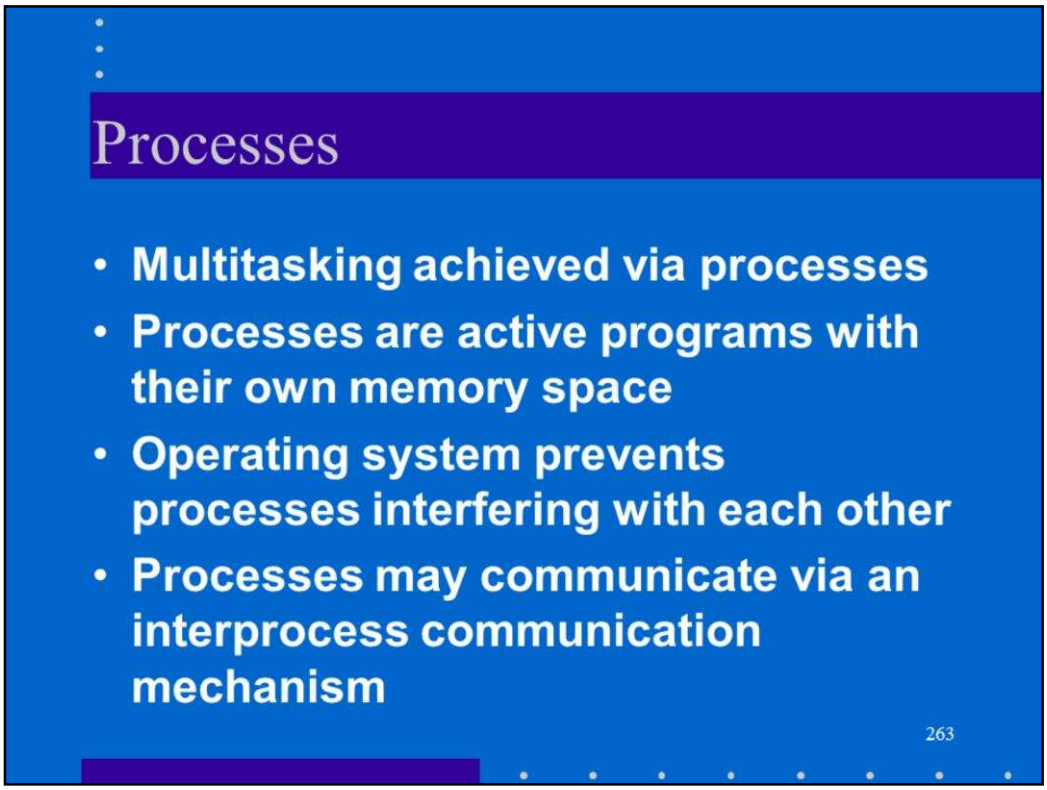
In this environment a user can type away quite happily at their keyboard and the operating system software deals with their input as fast as possible. Whenever the user pauses the operating system can switch to another task. When the user begins to type again the operating system switches back to the original task.

This is an oversimplification of what really goes on, but it is a good analogy of what happens in multitasking.


Types of multitasking
• Co-operative multitasking
– also known as non-pre-emptive
– Windows 3.1
– programs responsible for giving up processor
• Pre-emptive multitasking
– Windows '95 (32 bit), 2000, Unix
– Operating system shares resources
– Harder to implement, better results
262

There are a number of methods of organising the way in which resources are allocated to programs. Windows 3.1 allocated resources to a program in the hope that the program would execute some instructions and then hand control back to the operating system, which could then schedule a different application. The scheme depends on the application co-operating and so is known as *co-operative multitasking*. In cases where the application entered an endless loop and never reached the code which handed control back to the operating system, the whole machine became locked up.

Unix, Windows NT, and 32 bit programs running under Windows '95 are scheduled using a technique called *pre-emptive multitasking*. In this technique the operating system allocates resources to an application. This will enable it to execute. Rather than wait for the application to give the resources up, the operating system is activated at certain time intervals and may take the resources back from the executing application and allocate them to another application that is waiting.

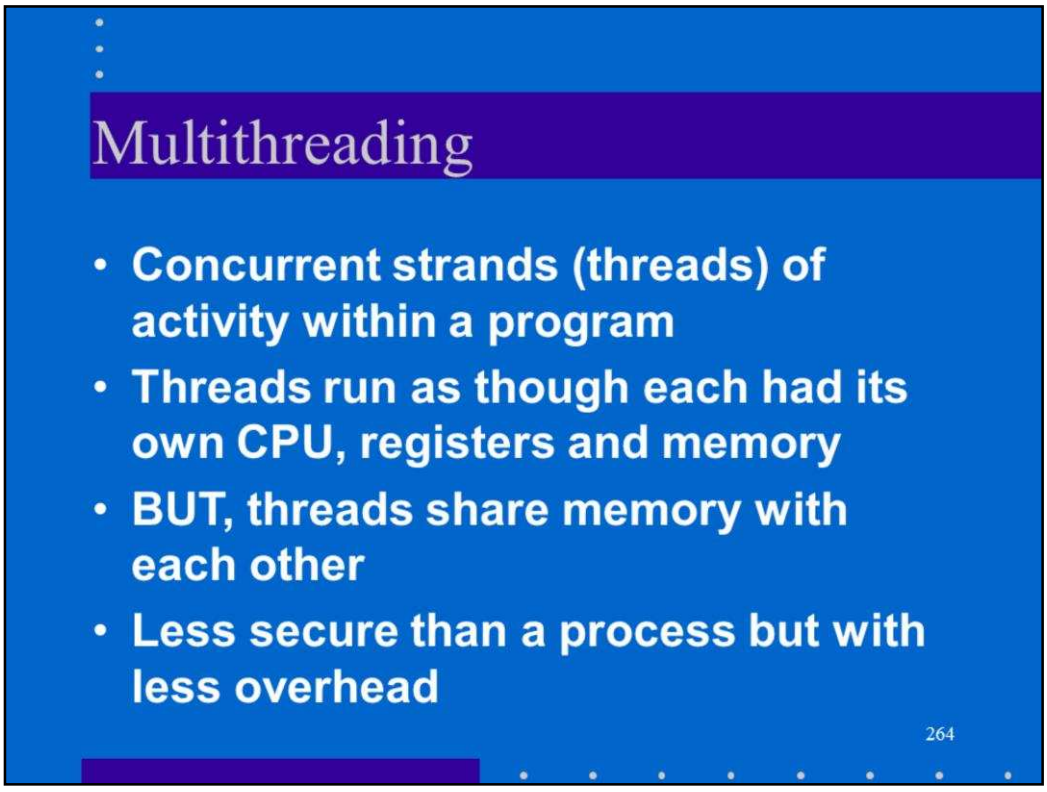

Processes
• Multitasking achieved via processes
• Processes are active programs with their own memory space
• Operating system prevents processes interfering with each other
• Processes may communicate via an interprocess communication mechanism
263

The description of multitasking that has been given so far is one that takes place at the process level. The operating systems decides whether Word or Excel or whatever should be the currently executing process.

A process can be thought of as an active program. Whilst it is active it has a state which will include such things as the next instruction to execute and current data values within the memory of the computer.

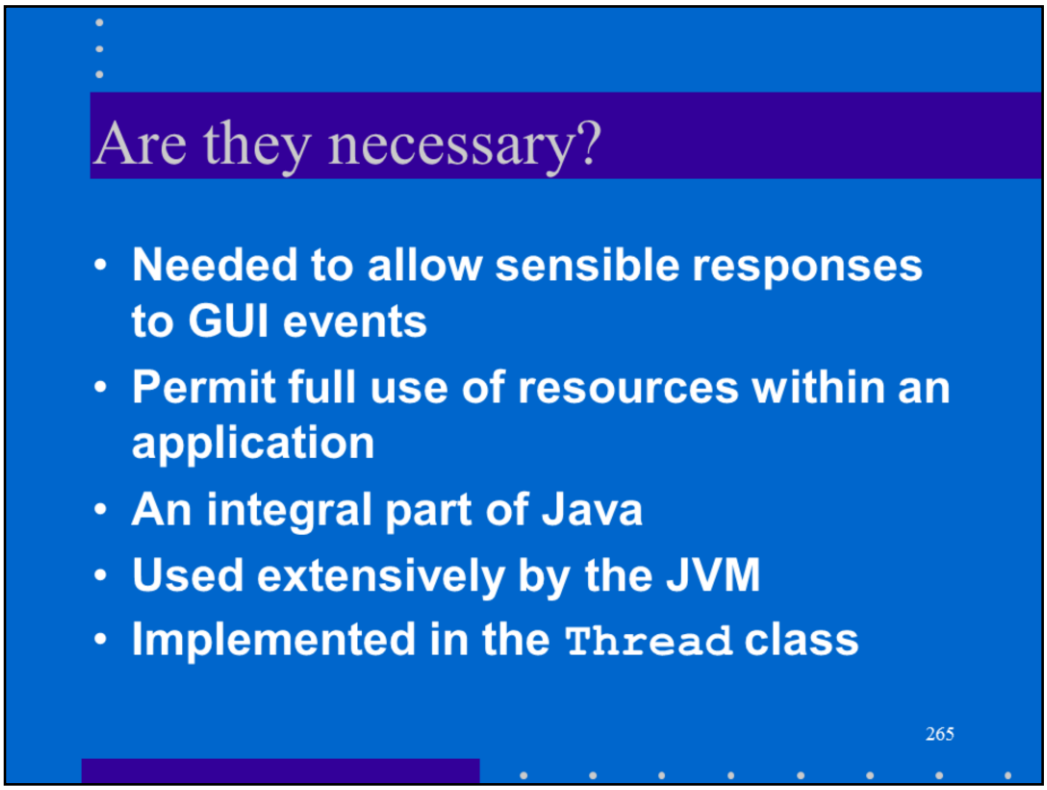
Operating systems try to keep processes from changing each others' states. If the variables created by Process A are changed, then it should be Process A that has made the changes and not any other process in the machine. This is a bit like the idea of encapsulation in O-O programming. Some operating systems are better than others at keeping processes apart! Windows NT is allegedly much better than Windows '95 at preventing one process from crashing another. This is why, although it has an almost identical interface to Windows '95, it is a separate and different product.

Since processes are deliberately kept apart, operating systems also supply secure communication mechanisms for those processes that wish to co-operate.


Multithreading
• Concurrent strands (threads) of activity within a program
• Threads run as though each had its own CPU, registers and memory
• BUT, threads share memory with each other
• Less secure than a process but with less overhead
264

Threads are similar in concept to processes, but whereas a process is normally created for a completely separate application, threads run within an application. Most GUI applications are required to perform a number of concurrent tasks and the thread mechanism enables this. Threads are similar to processes in that each thread behaves as though it has complete control of the CPU, registers and memory. The difference is that, whereas operating systems normally manage processes so that one process cannot access another process's memory; threads share memory with all the other threads in the application. For this reason, threads are sometimes known as lightweight processes.

Since threads share memory with other threads it is much easier for a thread to interfere with the operation of another thread, than it is for a process to interfere with another process. Conversely, however, it is much easier for threads to communicate with each other than it is for processes. For this reason, GUI applications are implemented as a collection of threads rather than as a collection of full blown processes.


Are they necessary?
• Needed to allow sensible responses to GUI events
• Permit full use of resources within an application
• An integral part of Java
• Used extensively by the JVM
• Implemented in the Thread class
265

For many years concurrency has been taught on computer science courses as a piece of theory which is rarely realised in practice. Concurrency mechanisms formed part of examinations but rarely coursework. It was clearly relevant to students specialising in operating systems but not part of the toolbox of the average applications programmer. One might, therefore, question whether multithreading should appear in a non-specialist programming course. The advent of GUIs however, has changed the nature of the programming task. Users expect concurrency in their applications. For example, it is perfectly reasonable to write an animation program which enables the user to pause the action. Such an application might consist of a window in which the action takes place and a button marked 'Pause'. If the only thread of execution is dedicated to carrying out the steps necessary to produce the graphical output then pressing the pause button will have no effect because that event will only be serviced when the program exits from its drawing code. What is required is one thread to do the drawing and another thread to check whether the pause button has been pressed. These threads must execute concurrently and must be able to interact.

Threads are essential elements of Java and are used extensively within the Java virtual machine. For example, garbage collection is undertaken by a separate thread which clears up the memory whenever a program is halted for input/output etc.

The JDK comes with a `Thread` class to enable multithreading.

Example(1)

```
class ThreadExample{
    private int count=0;

    class AddThread extends Thread{
        public void run(){
            for(int i=0;i<10;i++){
                count++;
                System.out.print("Add ");
                System.out.println(count);
            }
        }
    }
    .
    .
    .
```

266

Here is a class `ThreadExample` which has a single instance variable `count`. On creation of an instance of `ThreadExample`, `count` is set to zero.

Within `ThreadExample`, two inner classes `AddThread` and `SubtractThread` are declared. Each of these classes inherits from the `Thread` class. `Thread` contains a number of methods that may be overridden by a subclass. All subclasses of `Thread` will override the `run()` method which defines what the thread will do.

Class `AddThread` overrides `run()` so that it loops ten times and each time adds 1 to the `count` variable.

If `AddThread` were to run by itself it would produce output like this:

```
Add 1
Add 2
Add 3
Add 4
Add 5
Add 6
Add 7
Add 8
Add 9
Add 10
```

Example(2)

```
class SubtractThread extends Thread{
    public void run(){
        for(int i=0;i<10;i++){
            count--;
            System.out.print("Subtract ");
            System.out.println(count);
        }
    }
}
```

267

The second inner class `SubtractThread` overrides `run()` so that it subtracts one from `count` ten times.

If `SubtractThread` were to run by itself then it would produce the following output:

```
Subtract -1
Subtract -2
Subtract -3
Subtract -4
Subtract -5
Subtract -6
Subtract -7
Subtract -8
Subtract -9
Subtract -10
```

Example(3)

```
    .
    .
    public static void main(String args[]){
        ThreadExample ex = new ThreadExample();
        AddThread a = ex.new AddThread();
        SubtractThread b = ex.new SubtractThread();
        a.start();
        b.start();
    }
}
```

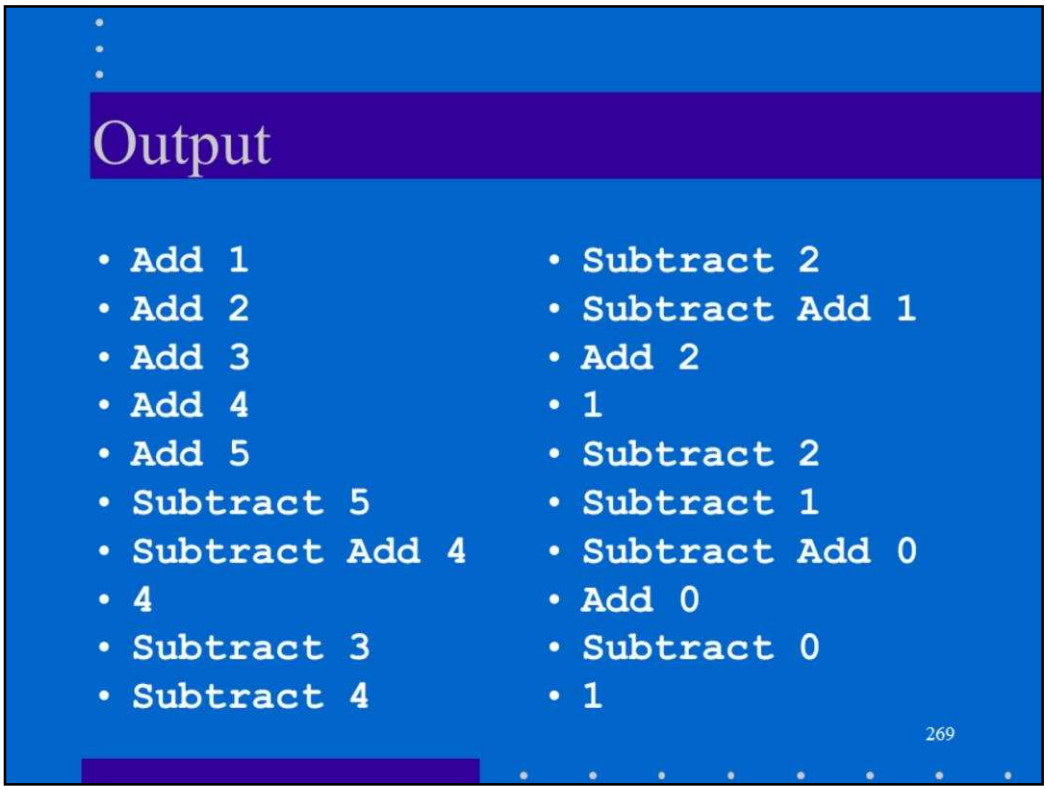
268

The `main()` method of the application begins by creating an instance of class `ThreadExample`. In the next two lines it creates instances of `AddThread` and `SubtractThread`. Since `AddThread` and `SubtractThread` are inner classes of `ThreadExample`, instances of these classes must be associated with an instance of `ThreadExample`. Therefore, since variable `ex` contains a reference to an instance of a `ThreadExample`, the syntax `ex.new` is used to indicate that the instance of the inner class created will be associated with the newly created instance (referenced by `ex`) of `ThreadExample`.

Variable `a` is set to refer to an instance of `AddThread` and variable `b` is set to refer to an instance of `SubtractThread`.

Once created, a thread does not do anything until it is instructed to execute. This is achieved via a method of the Thread class: `start()`.

The code `a.start()` initiates a separate thread which executes the code in the `run()` method of the `AddThread` class. This will run concurrently with the thread in which the `main()` method of `ThreadExample` is executing. Consequently `ThreadExample` can start another thread before the `AddThread` thread has finished. The statement `b.start()` kicks off a third concurrent thread which this time executes the code in the `run()` method of class `SubtractThread()`.



The output from this requires some explanation! We have two threads, A which implements the `run()` method of `AddThread` and B which implements the `run()` method of `SubtractThread`. The processes run concurrently and either of them may have control of the processor at any time.

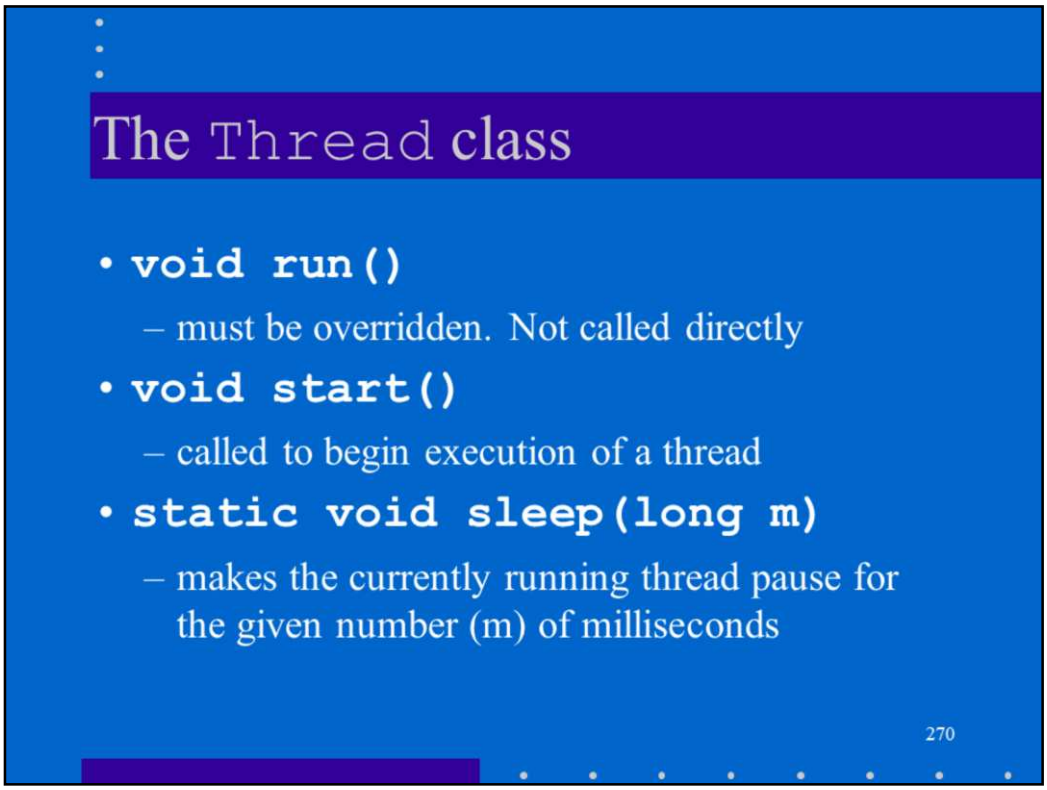
A runs first and loops round successfully five times. It loops a sixth time and adds 1 to `count`, so that `count` has a value of 6. Before A can print this value out, the processor is handed over to B which subtracts 1 from the current value of `count` giving 5. B prints its result out. B loops round again subtracts 1 from `count` giving 4 and begins to print out its message. It gets as far as printing out the string "Subtract" when A is handed the processor. A continues from where it left off previously and prints the string "Add" followed by the current value of `count` which is now 4. B is then handed the processor and finishes off the output it started before by printing out the current value of `count`, now 4.

This goes on in a random pattern but at the end the string "Add" will be printed ten times, the string "Subtract" will be printed ten times and the value of `count` will be printed twenty times.

The output from the program will vary from execution to execution depending on what other things the processor is doing on each occasion.

This looks chaotic but it does demonstrate that the threads are executing concurrently. Later in the lecture, we will see how thread execution can be

synchronised to produce more predictable results.


The Thread class
• void run()
– must be overridden. Not called directly
• void start()
– called to begin execution of a thread
• static void sleep(long m)
– makes the currently running thread pause for the given number (m) of milliseconds
270

The `ThreadExample` class makes use of class `Thread`. `Thread` like most Java classes contains a number of methods (see Java API documentation for full details).

All classes inheriting from `Thread` must override the method `run()`. This method should not be called directly from an application or applet. It is invoked by the JVM when the thread is executed.

The `start()` method is called to begin execution of a thread. The method requests the JVM to schedule the thread.

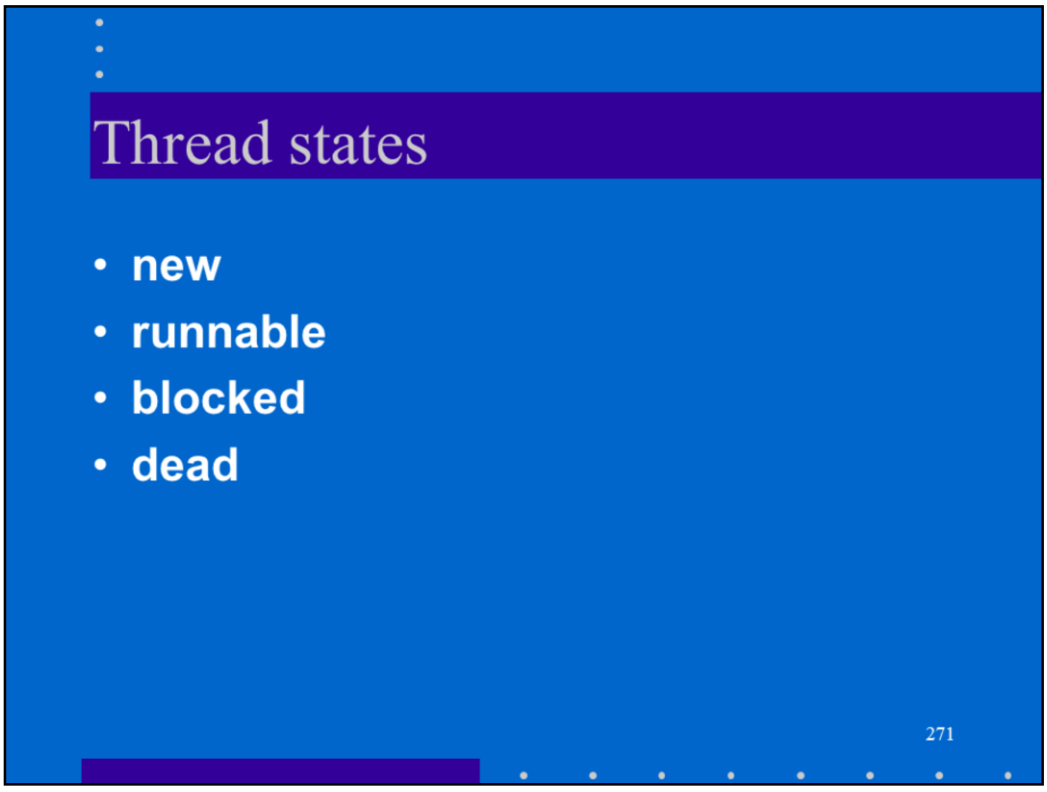
The `sleep()` method enables a thread to do nothing for a set period of time. This can be very useful if the thread is waiting for another thread to finish an action. Alternatively, it might be used to wait for an external event such as a message arriving from another computer.

The following methods were (for technical reasons) necessary in thread programming in early versions of the JDK. They are DEPRECATED in Java 2 and should not be used, however you will still see them in text books.

The `stop()` method can be used to terminate a thread.

The `suspend()` method can be used to pause the execution of a thread.

The `resume()` method is used to resume execution of a suspended thread.



In their lifetimes threads go through a number of states. The state of a thread determines whether the thread will be allocated resources by the JVM.

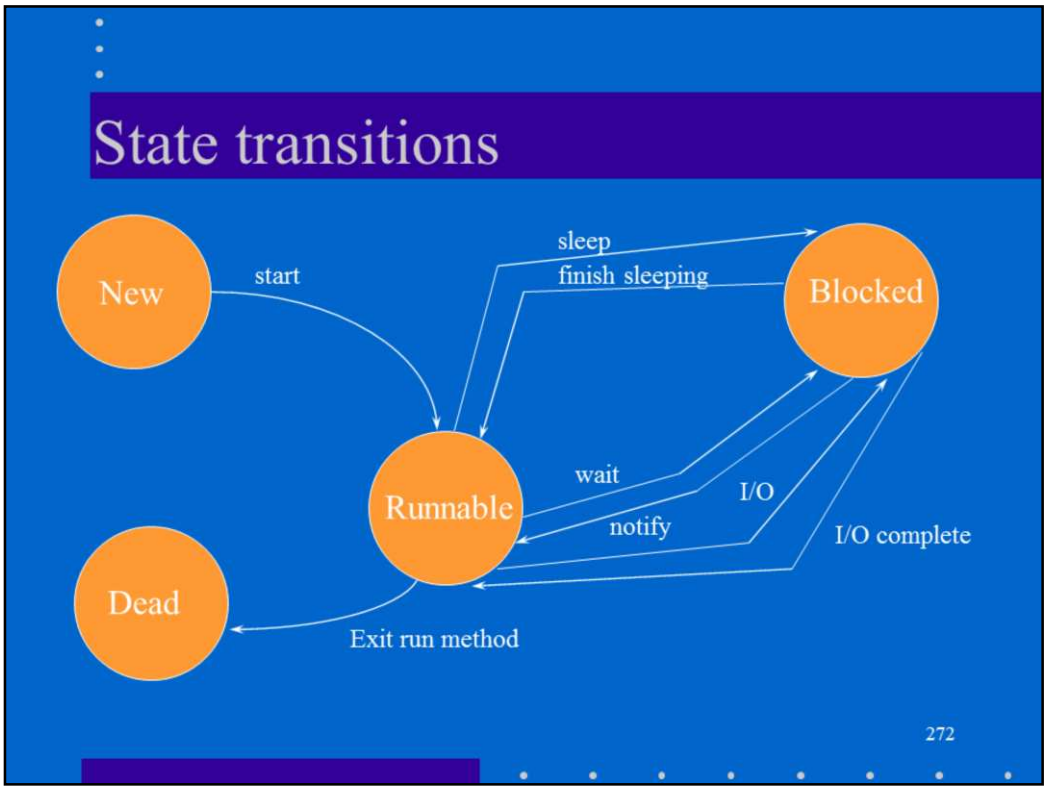
The four states a thread may have are:

new

runnable

blocked

dead



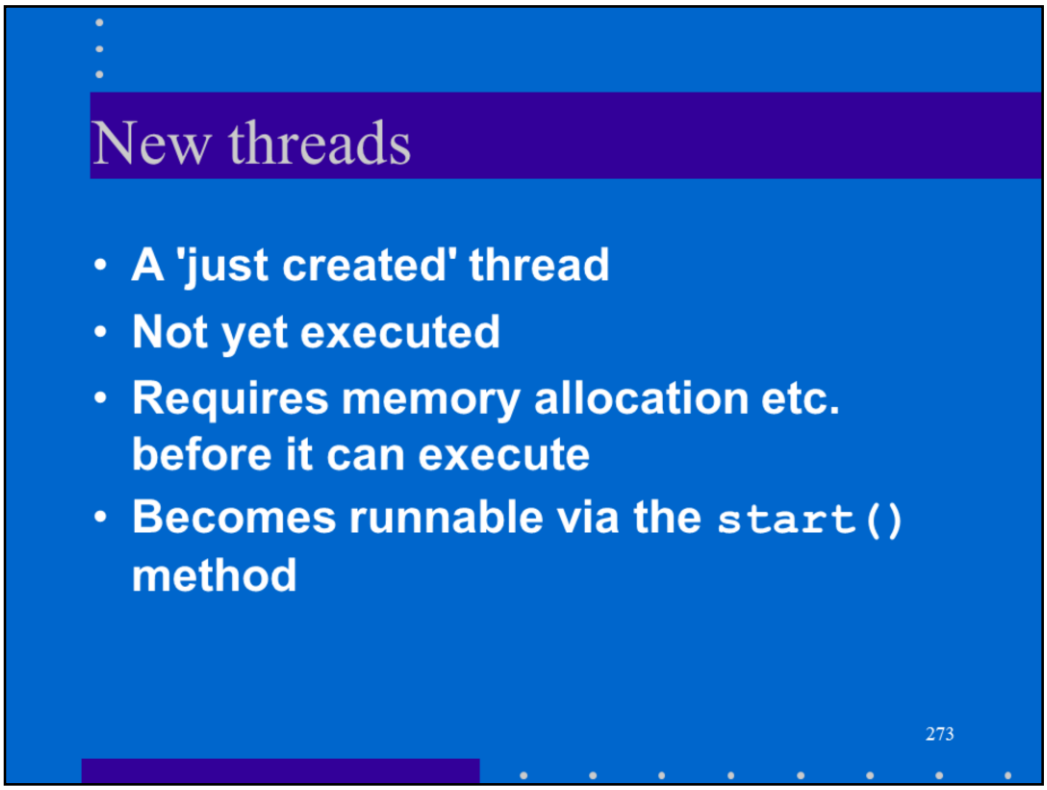
This diagram shows how a thread may change state. A thread in the New state becomes Runnable when its `start()` method is invoked. From the Runnable state there are a number of events which will place the thread into the Blocked state.

A thread can become Runnable again if an appropriate event occurs when it is in the Blocked state. For example, if a thread sleeps for 10 milliseconds it will become blocked for 10 milliseconds after which it will return to the runnable state.

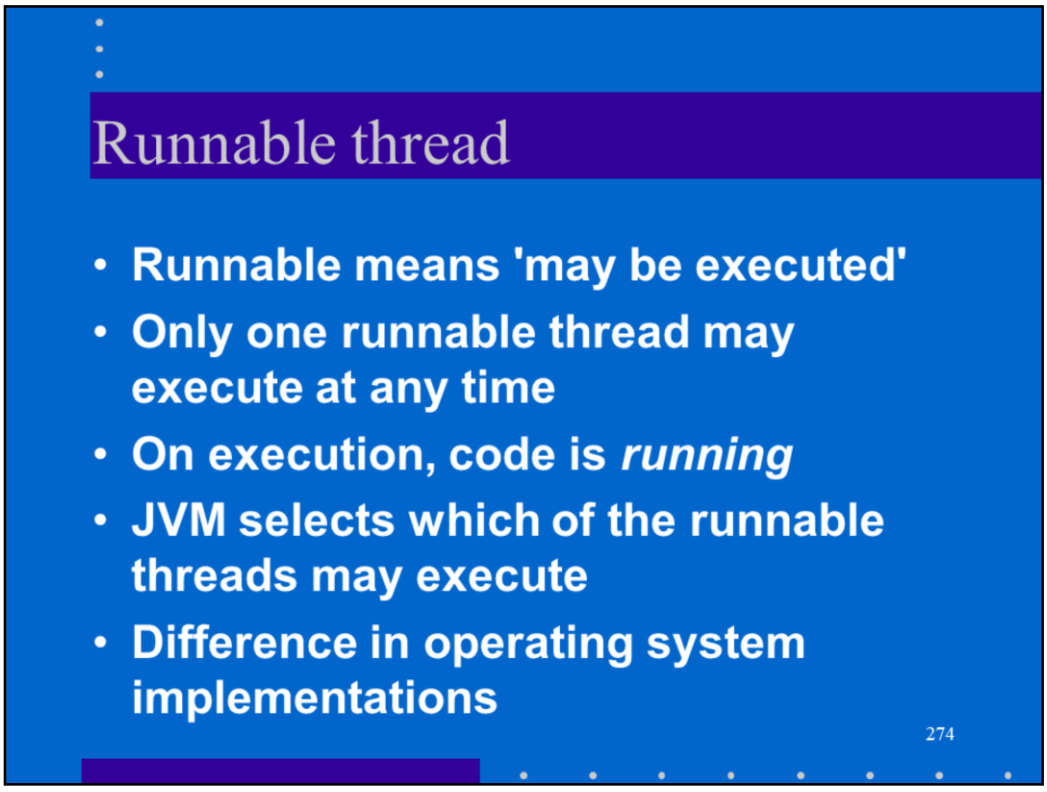
Threads only move from being Blocked to being Runnable when an event occurs that is related to the event that caused them to be Blocked. For example, if a thread is waiting for I/O, it can only become runnable when that I/O is complete. It cannot become runnable via a call to `resume()`.

Threads are only eligible for execution when they are Runnable. The JVM selects one thread from the set of Runnable threads to run.

A Runnable thread becomes Dead when iit returns from its `run()` method. A Dead thread can never execute again.

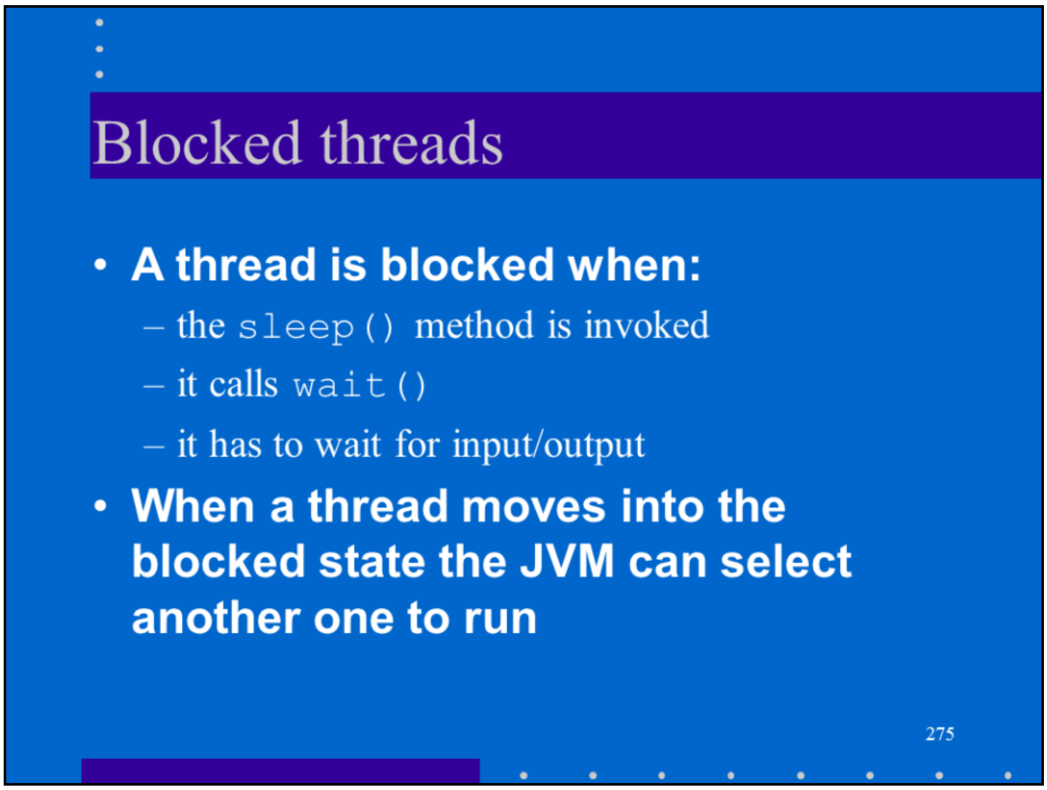

New threads
• A 'just created' thread
• Not yet executed
• Requires memory allocation etc. before it can execute
• Becomes runnable via the start() method
273

A thread in the new state is one that has been created using the `new` operator but has not been started. Invoking the `start()` method for this thread will allocate memory to the thread and make it runnable.


Runnable thread
• Runnable means 'may be executed'
• Only one runnable thread may execute at any time
• On execution, code is *running*
• JVM selects which of the runnable threads may execute
• Difference in operating system implementations
274

When a new thread has it start() method called it becomes runnable. That is to say, it joins the list of threads which the JVM may choose to execute. When the JVM has selected a particular thread, that thread actually executes. There is no way to force the JVM to choose a given thread but it is possible to influence its choice.

This is one point in Java where there is a difference between implementations of the different operating systems. It is possible under some Solaris (Unix) implementations of Java for a thread to run for ever. Windows NT and Windows 95 will let a thread run for a certain amount of time and then choose to run a different thread from the set of runnable threads. The Windows implementation is preferable and the latest versions of the Solaris implementionl behave more like the Windows version.


Blocked threads
• A thread is blocked when:
– the sleep() method is invoked
– it calls wait()
– it has to wait for input/output
• When a thread moves into the blocked state the JVM can select another one to run
275

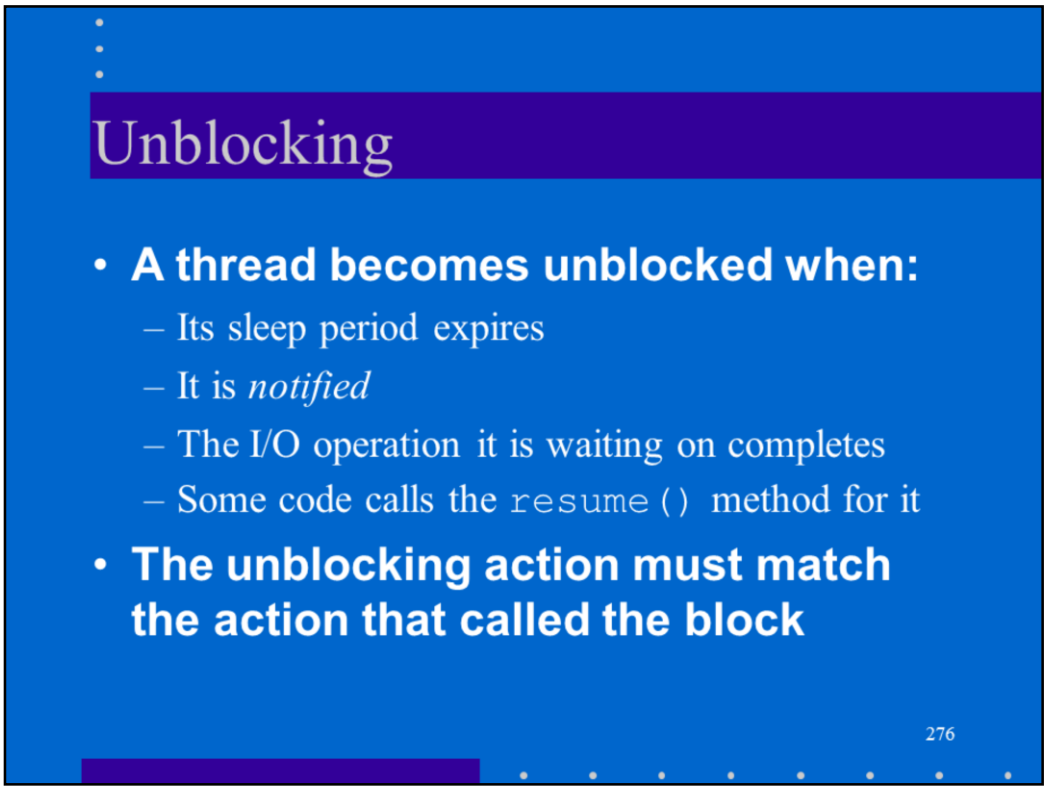
A thread that is runnable may move into the blocked state. This may be caused by the thread itself or by another concurrent thread. A thread may choose to wait for a specified time and invoke the `sleep()` method. It will become blocked for the time specified in the call to `sleep()`.

A blocked thread will not be selected for execution by the JVM.

A thread may want to use a shared resource. Such a resource may be used by a number of threads but by only one at time. A thread which wishes to make use of a shared resource which is already in use will call `wait()`. This will cause the thread to be blocked. The thread will become runnable again when the resource becomes available.

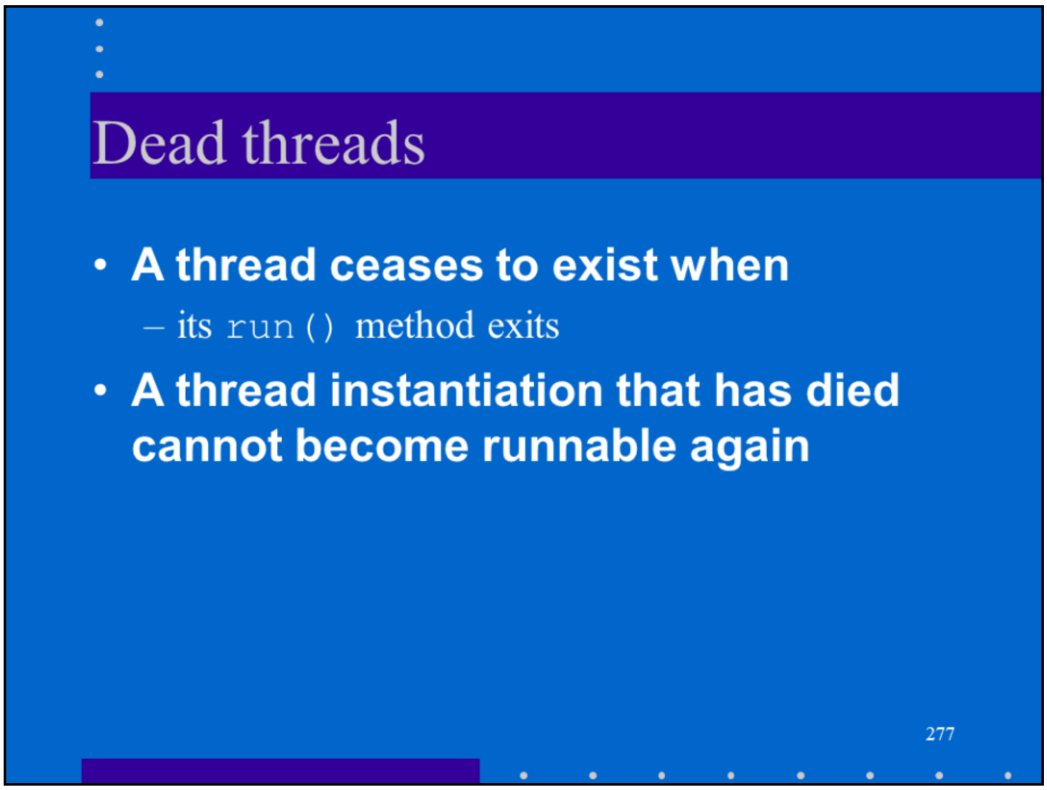
Threads become blocked as soon as they undertake input or output. This is because the processor can operate many times faster than any of the input or output devices. If the processor had to wait while an I/O operation took place it would waste a lot of processor power. The thread will become runnable again when the input or output operation has completed.

When a running thread is blocked the JVM selects another thread from the list of runnable threads to execute. This makes the best use of the processor time.


Unblocking
• A thread becomes unblocked when:
– Its sleep period expires
– It is notified
– The I/O operation it is waiting on completes
– Some code calls the resume() method for it
• The unblocking action must match the action that called the block
276

Threads move out of the blocked state into the runnable state when the reason for blocking it is no longer valid.

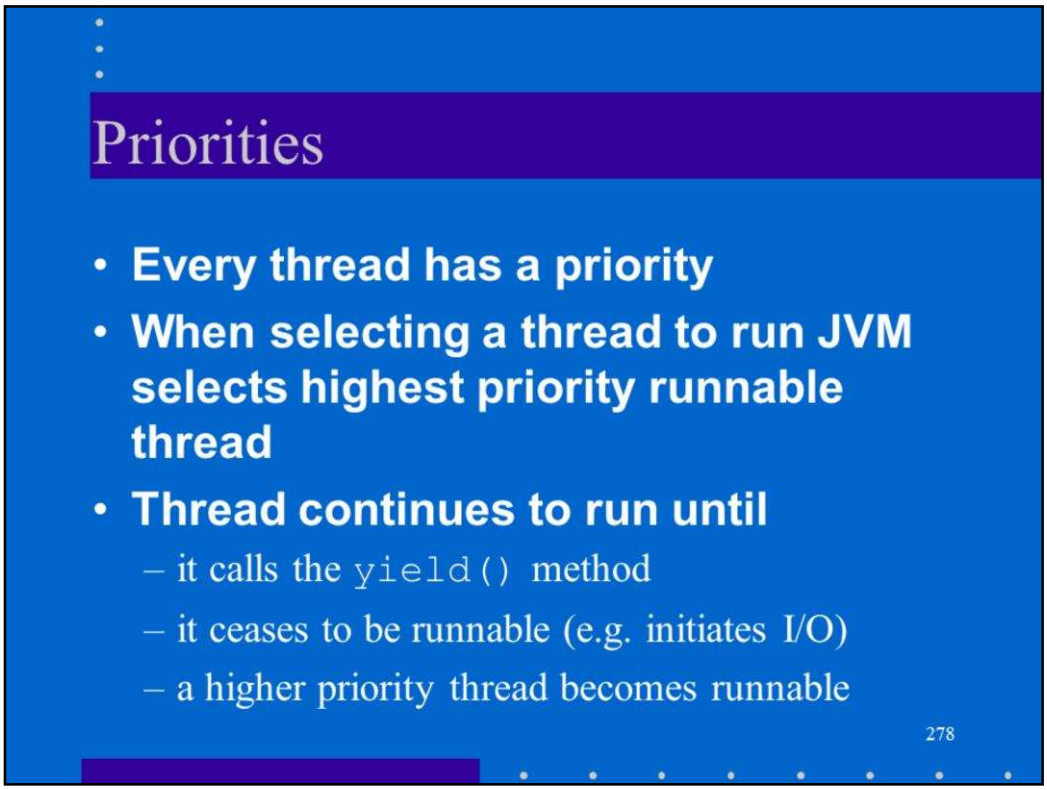
Note that `suspend()/resume()` is now deprecated in Java 2. The reason for this is that the use of `suspend()` and `resume()` often leaves the objects that a thread operates on in a inconsistent state. As of Java 2, use of `wait()` and `notify()` or `notifyall()` is preferred.


Dead threads
• A thread ceases to exist when
– its run() method exits
• A thread instantiation that has died cannot become runnable again
277

Threads may die. A dead thread still holds some resources and eventually these will be cleared up by the system. This is achieved by a mechanism similar to the exception mechanism (ThreadDeath) and is not normally the responsibility of the programmer.

Threads terminate either when they return from the `run()` method or else the thread is explicitly killed using the `stop()` method.

`stop()` is now deprecated and threads should terminate by returning from `run()`.


Priorities
• Every thread has a priority
• When selecting a thread to run JVM selects highest priority runnable thread
• Thread continues to run until
– it calls the yield() method
– it ceases to be runnable (e.g. initiates I/O)
– a higher priority thread becomes runnable
278

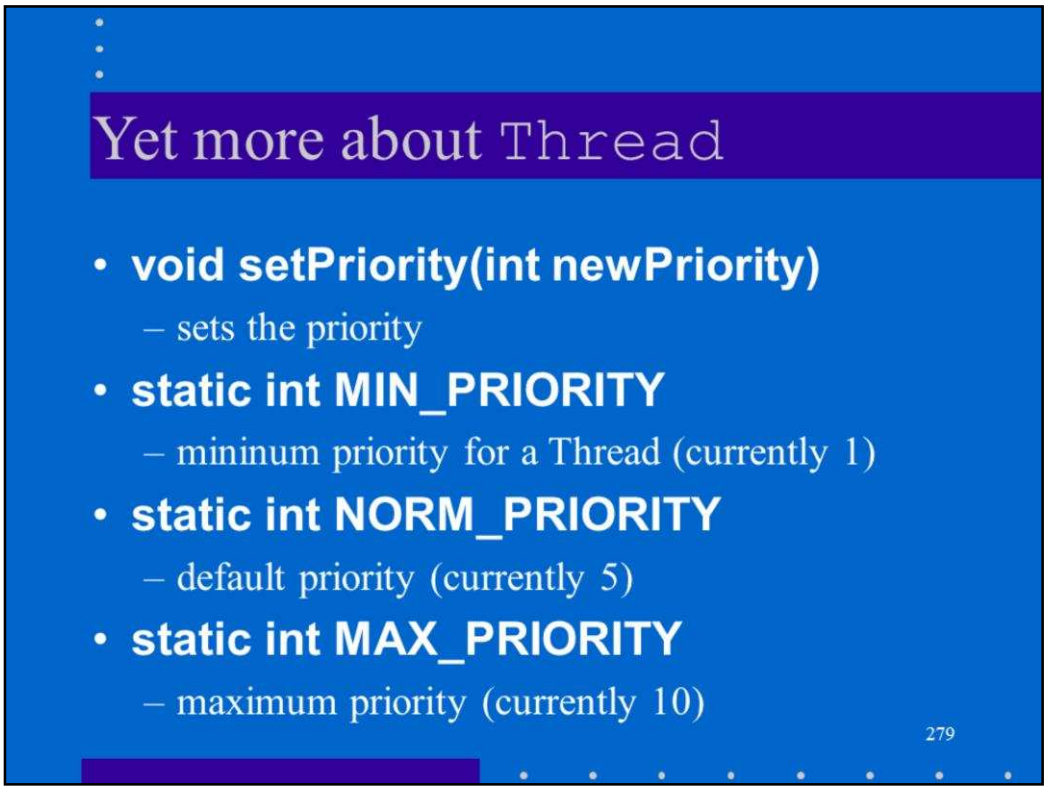
The JVM does not select a thread to run on a purely random basis. All threads are assigned a priority. When the JVM selects a thread to execute it will choose from the runnable threads with the highest priority.

Once selected, a thread will continue to run without interruption until it moves into the blocked state (which might result because the thread does some I/O) or a higher priority thread becomes runnable or until it decides to give up the processor to another thread. This last option is taken by calling the `yield()` method.

```
public static native void yield()
```

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

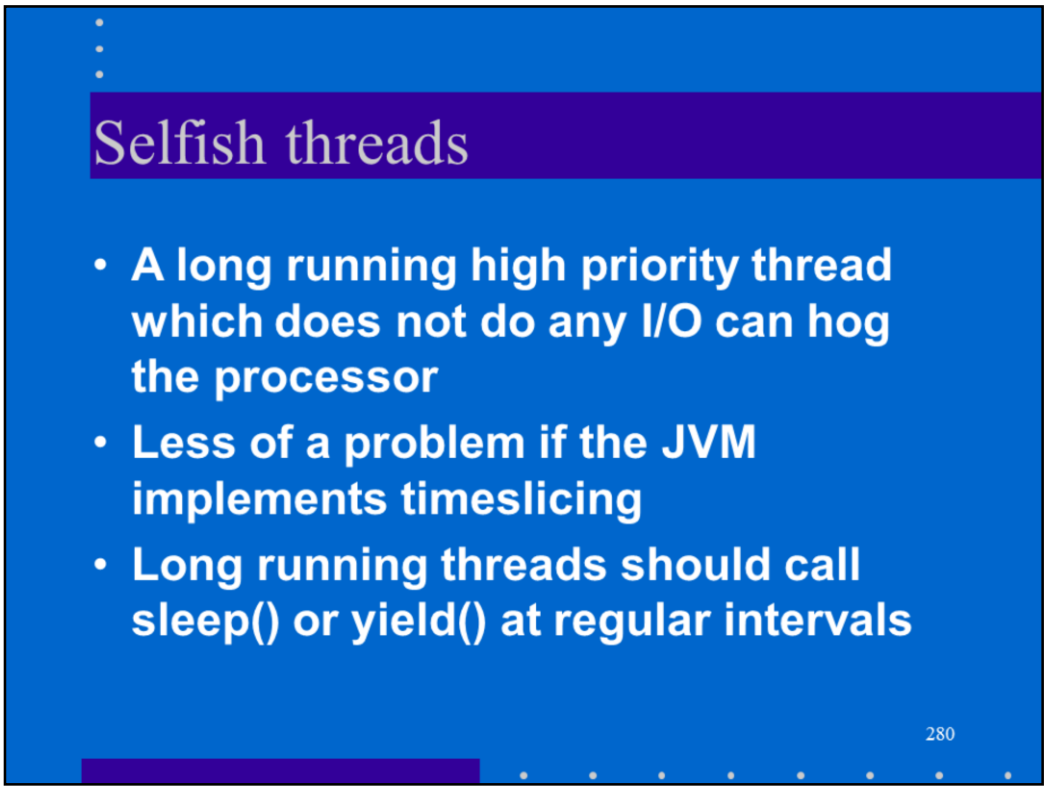
The above is not in fact true for Windows as the Windows operating system allocates a time slot to each running thread. When the end of the slot is reached, Windows will choose a new thread from the set of runnable threads. On the other hand, for the sake of portability, it is wise to behave as though it were true since whilst a thread might behave itself on Windows, it could hog the processor in a Unix environment.


Yet more about Thread
• void setPriority(int newPriority)
– sets the priority
• static int MIN_PRIORITY
– mininum priority for a Thread (currently 1)
• static int NORM_PRIORITY
– default priority (currently 5)
• static int MAX_PRIORITY
– maximum priority (currently 10)
279

The `Thread` class contains a method `setPriority()` to set the priority at which a thread runs.

There are three class variables of `Thread` which contain values for the maximum priority possible, the minimum value possible and the default (normal) value for a priority. Currently the lowest possible priority is 1 and the highest is 10. It would, however be unwise to rely on these values remaining the same in future implementations of Java.

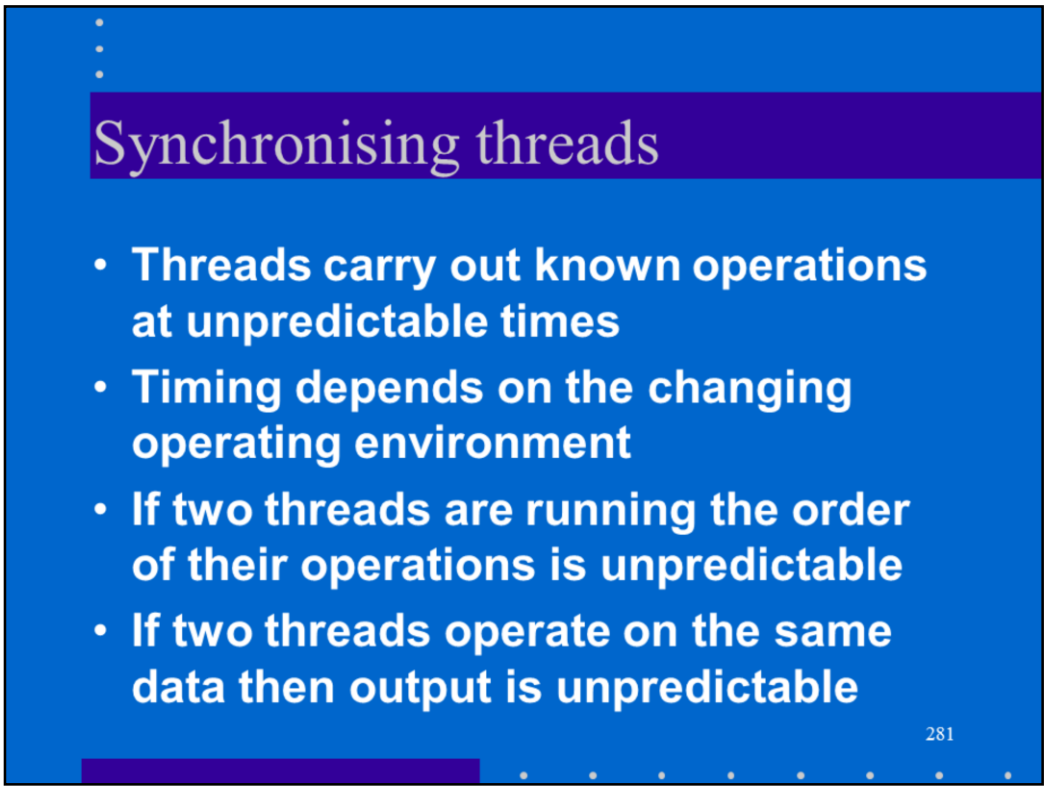
A thread which has a higher priority is likely to receive more processor time than one at a lower priority. The speed at which a thread executes will, however, also depend on how often it enters the blocked state.


Selfish threads
• A long running high priority thread which does not do any I/O can hog the processor
• Less of a problem if the JVM implements timeslicing
• Long running threads should call sleep() or yield() at regular intervals
280

Polite threads will always call `yield()` or `sleep()` at appropriate points in order to allow other threads to execute.

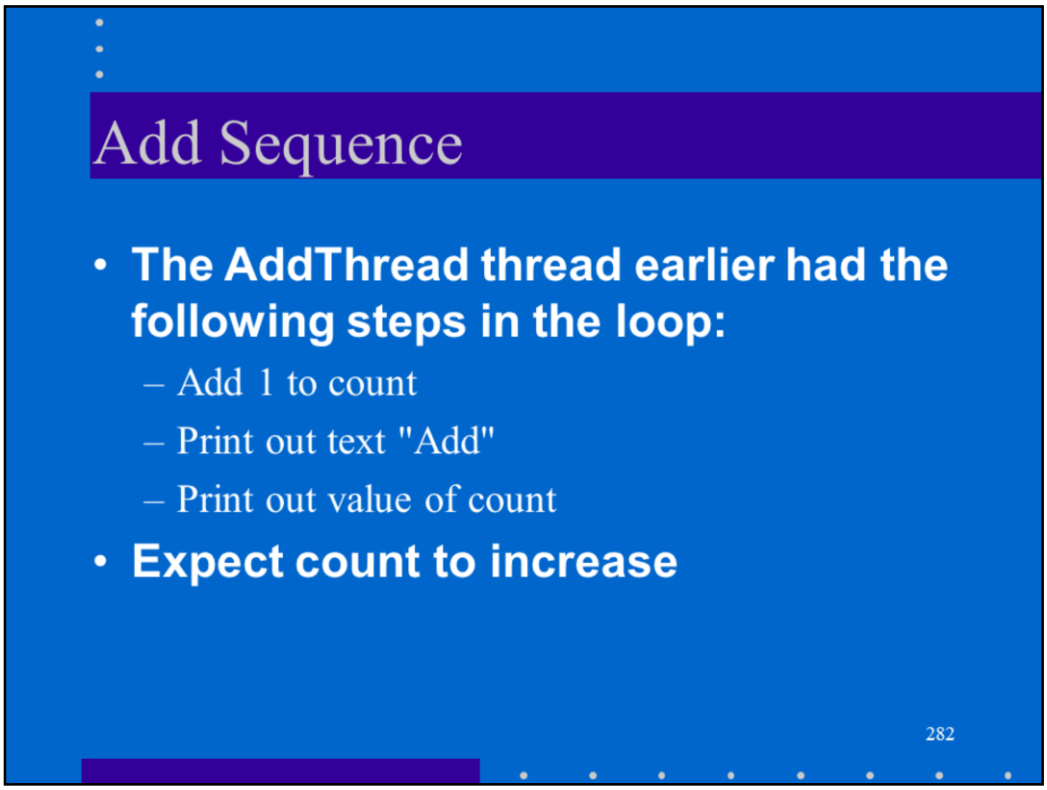
A thread running at the highest priority is also likely to hog the processor and prevent other threads from executing.

This is less of a problem if the JVM implements timeslicing (as it does in Windows) but if you are writing a cross-platform application you cannot rely on this.

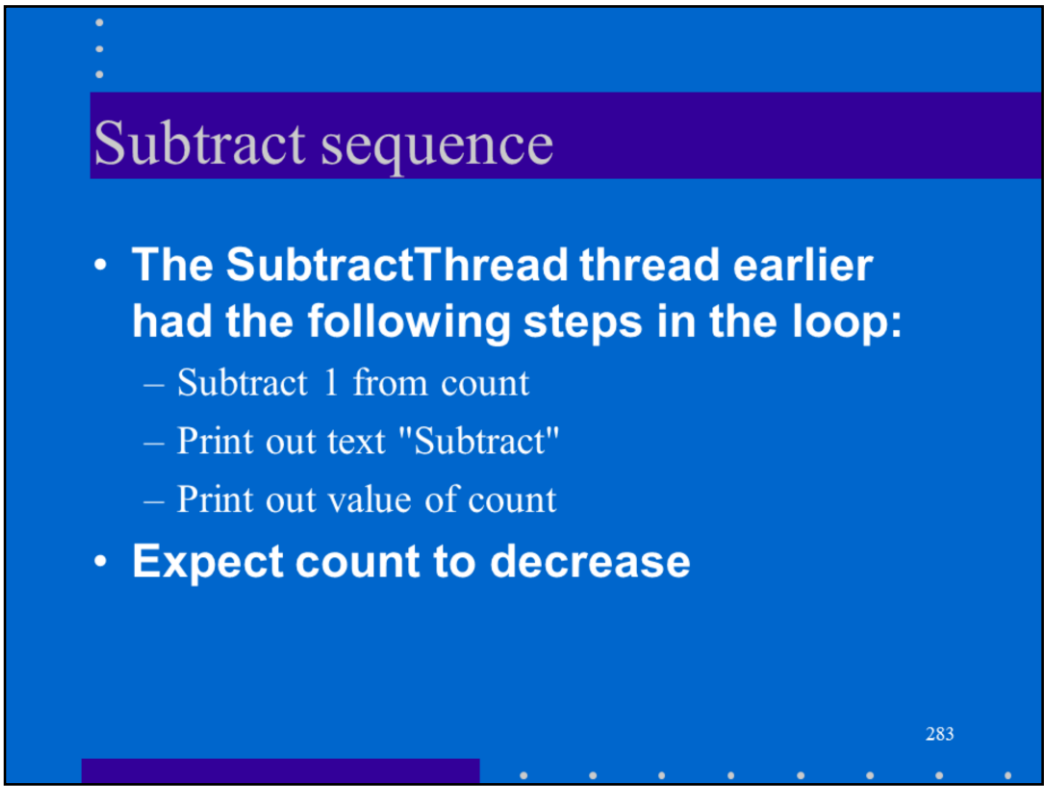

Synchronising threads
• Threads carry out known operations at unpredictable times
• Timing depends on the changing operating environment
• If two threads are running the order of their operations is unpredictable
• If two threads operate on the same data then output is unpredictable
281

From the description of threads you can see that it is impossible to accurately predict when a thread will execute and how long it will execute for. For instance, it is impossible to know how long any thread undertaking an I/O operation will be blocked as there may be other applications on the machine which are trying to simultaneously access the same device.

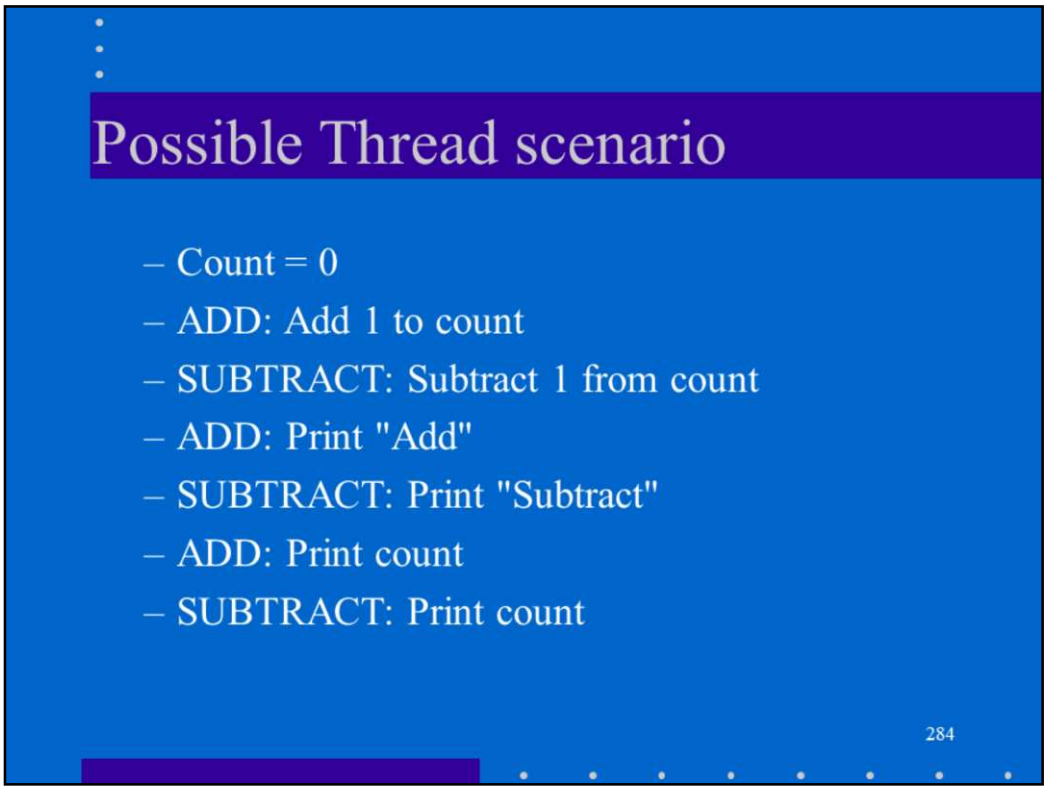
If two threads are operating on the same data then the results can be unpredictable if there is no mechanism which regulates their behaviour. On occasions thread A may execute before B, on other occasions thread B may execute before thread A. Worse than this, the operations of thread A may become mixed up with the operations of thread B. This is why the results of running the `ThreadExample` program shown earlier are so unpredictable.


Add Sequence
• The AddThread thread earlier had the following steps in the loop:
– Add 1 to count
– Print out text "Add"
– Print out value of count
• Expect count to increase
282

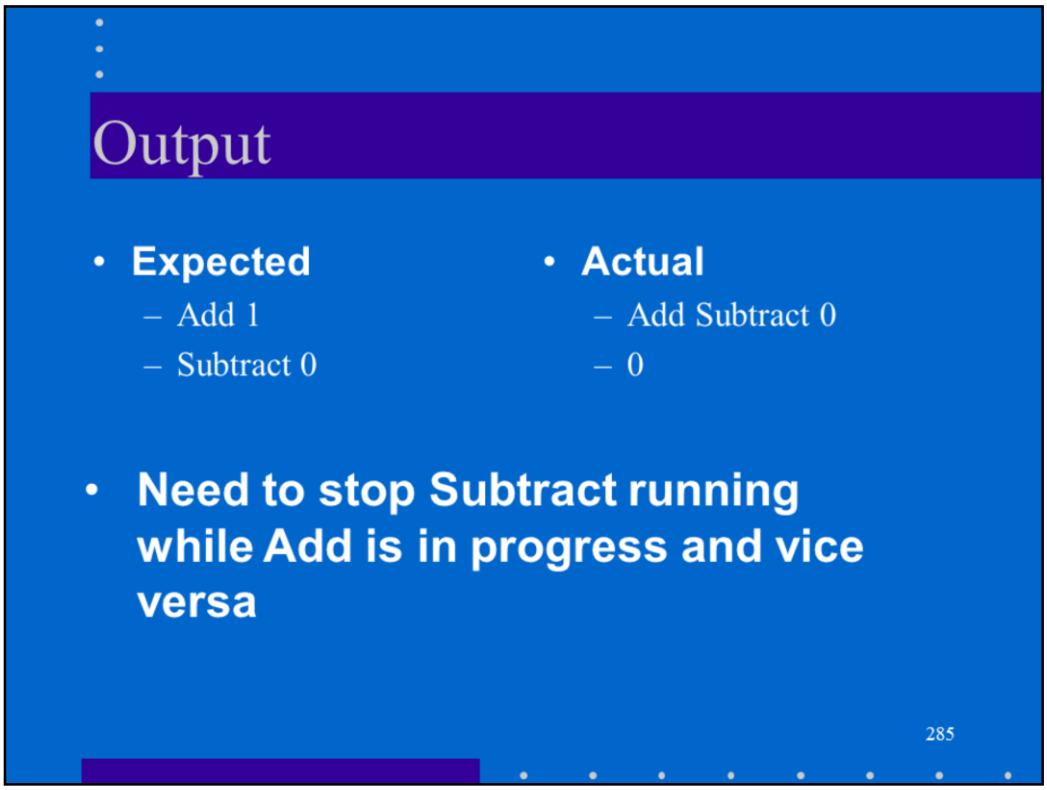
In the `ThreadExample` program, one of the threads `AddThread` contained three basic steps. At the end of an execution of this thread the expectation is that the `count` variable will grow larger.


Subtract sequence
• The SubtractThread thread earlier had the following steps in the loop:
– Subtract 1 from count
– Print out text "Subtract"
– Print out value of count
• Expect count to decrease
283

`SubtractThread` also has three basic steps. At the end of an execution of `SubtractThread` we expect the value of the `count` variable to decrease.


Possible Thread scenario
– Count = 0
– ADD: Add 1 to count
– SUBTRACT: Subtract 1 from count
– ADD: Print "Add"
– SUBTRACT: Print "Subtract"
– ADD: Print count
– SUBTRACT: Print count
284

This example shows an execution trace of the two threads. Remember that the JVM can choose to switch between the two threads at any time it feels. This means that the individual operations of `AddThread` and `SubtractThread` can become interleaved.



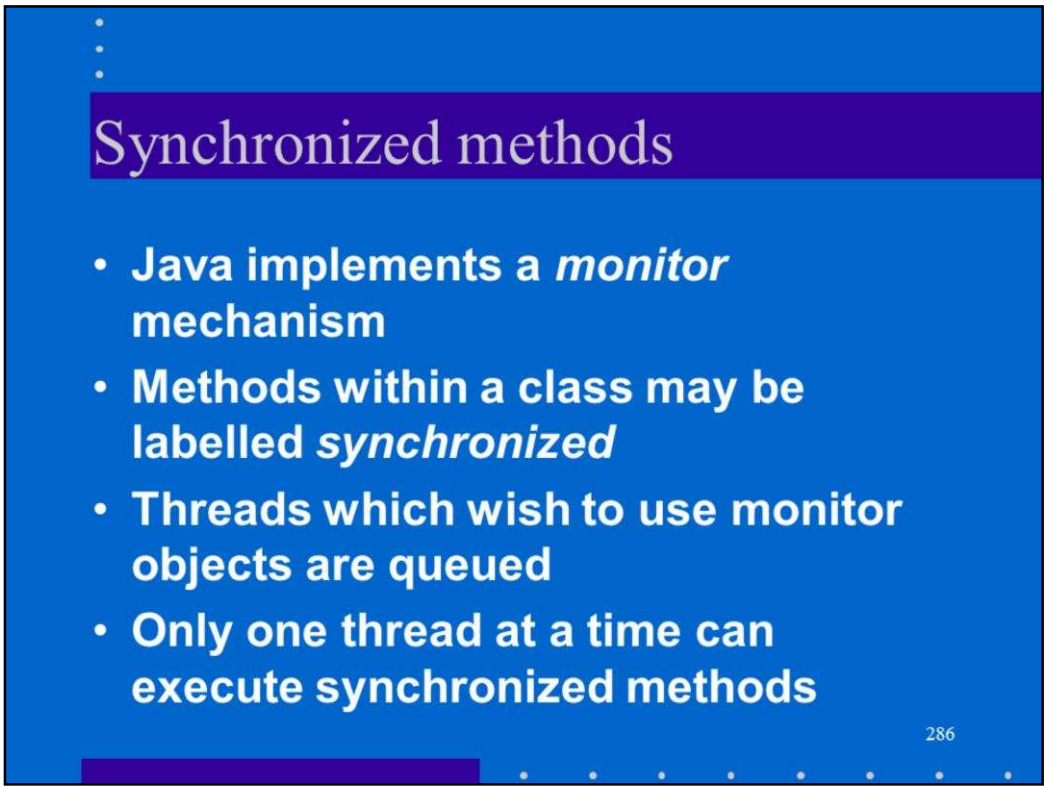
We might expect output in which each line begins with either Add or Subtract (but not both) and is followed by a number. We also might expect that the number on each line will be either one less than or one greater than the number on the line before.

The output of the sequence of operations on the previous slide is:

```
Add Subtract 0
0
```

The first line contains both Add and Subtract and the number on both lines is identical rather than differing by one.

To get a correct answer we need a mechanism whereby `AddThread` does not interfere with `SubtractThread`, ie the operations of `SubtractThread` are not allowed to start until `AddThread` has finished, and vice versa.


Synchronized methods
• Java implements a monitor mechanism
• Methods within a class may be labelled synchronized
• Threads which wish to use monitor objects are queued
• Only one thread at a time can execute synchronized methods
286

Java implements a technique to do this known as a monitor. Monitors are well known mechanisms in operating systems and were first described in a paper written by C. A. R Hoare entitled *Communicating Sequential Processes* (*Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666-677).

Java makes implementing monitors particularly easy. The modifier `synchronized` is added to a method definition. When a thread invokes a synchronized method on a particular object it automatically obtains a lock for that object. Any other thread which tries to invoke the same, or any other synchronised method for that object whilst the first thread holds the lock for the object must wait until the lock is released. The lock is only released when the first thread exits from the synchronised method.

Only one synchronised method can be executed at one time on a given object.

All the locking and queuing is looked after automatically by Java.

Example (1)

```
class SynchThreadExample{
   private int count=0;

   private synchronized void incCount(){
        count++;
        System.out.print("Add ");
        System.out.println(count);
   }

   private synchronized void decCount(){
        count--;
        System.out.print("Subtract ");
        System.out.println(count);
   }
   .
```

287

In this version of ThreadExample the code for incrementing and decrementing `count`, together with the code for printing out the result has been placed in the synchronized methods `incCount()` and `decCount()`.

Example(2)

```
.
.
class AddThread extends Thread{
    public void run(){
        for(int i=0;i<10;i++)incCount();
    }
}

class SubtractThread extends Thread{
    public void run(){
        for(int i=0;i<10;i++)decCount();
    }
}
.
.
```

288

The two threads simply call the synchronized methods.

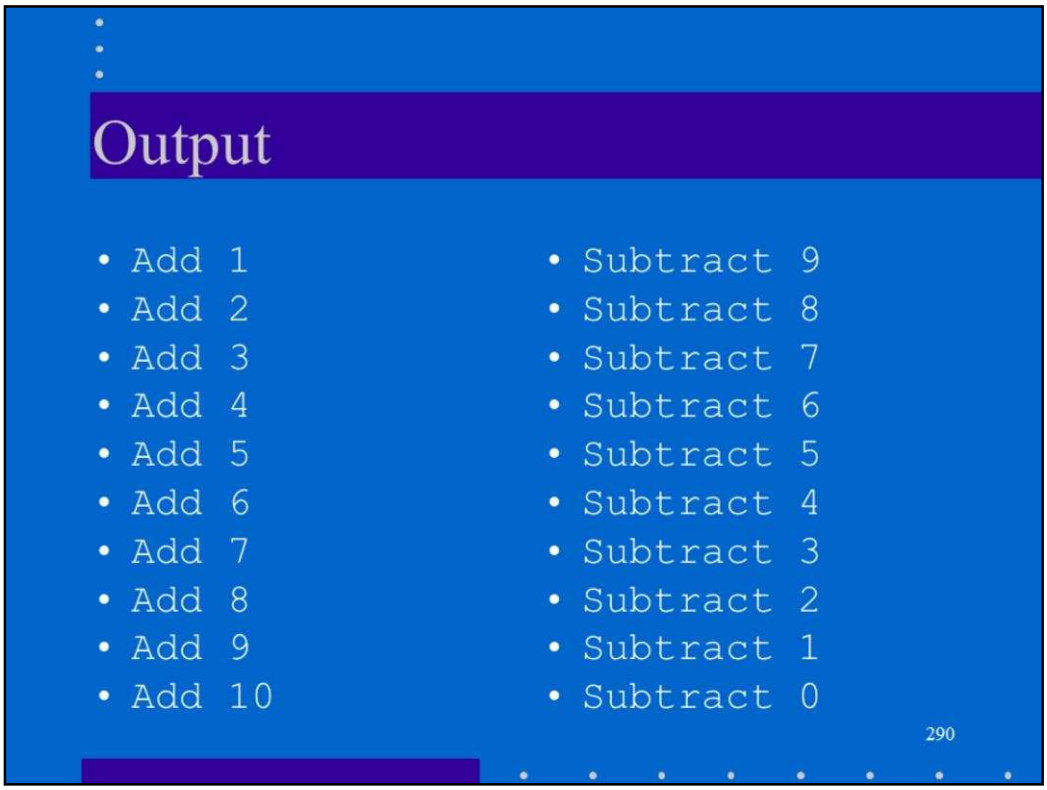
Note that methods can be synchronized but classes, even when they represent threads, cannot.

Example(3)

```
    .
    .
    .
    public static void main(String args[]){
        SynchThreadExample ex =
                        new SynchThreadExample();
        AddThread a = ex.new AddThread();
        SubtractThread b = ex.new SubtractThread();
        a.start();
        b.start();
    }
}
```

289

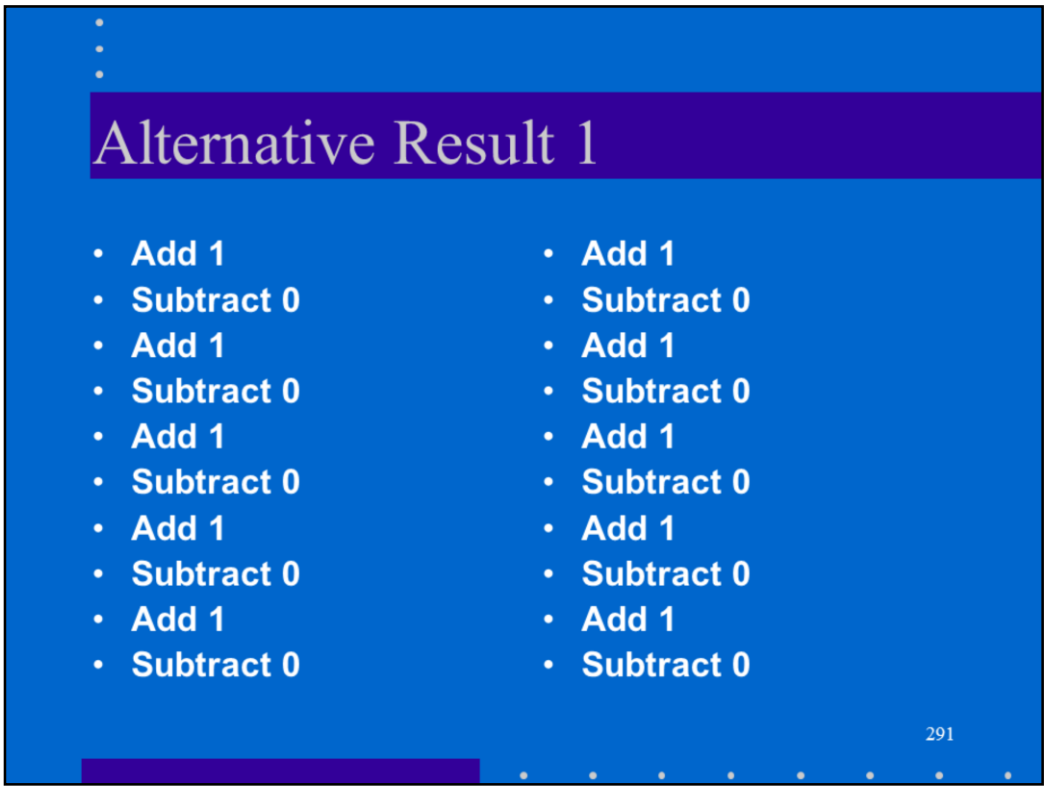
The `main()` method does not alter.



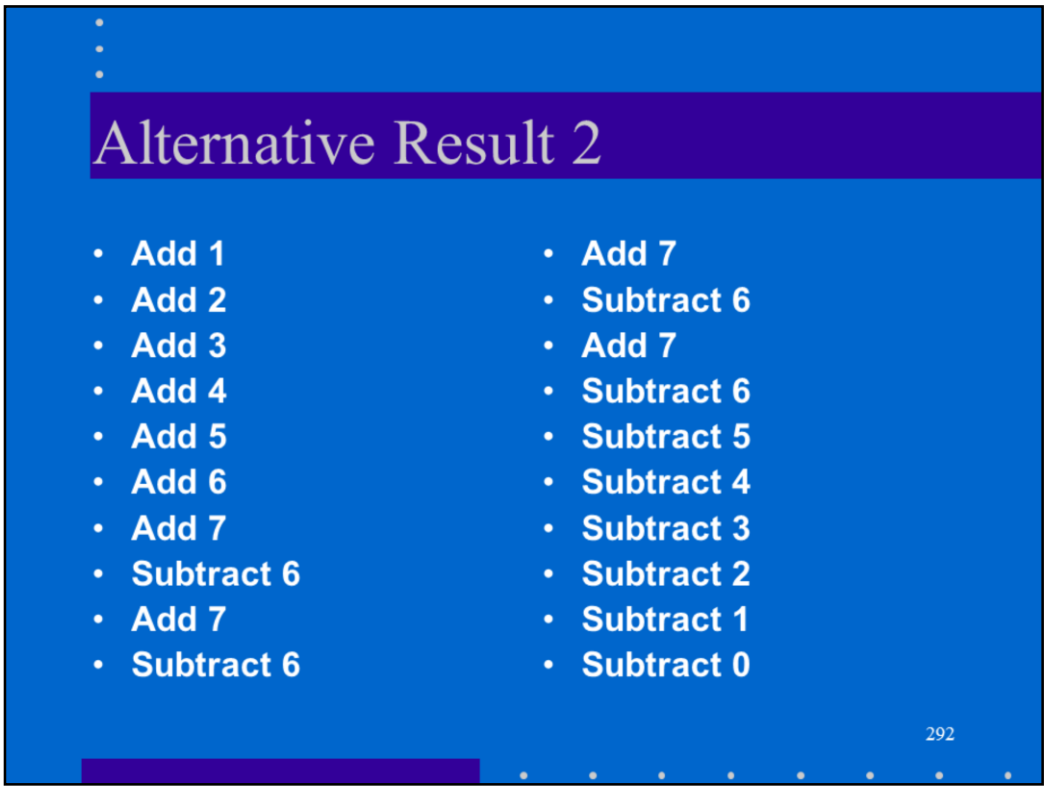
This is the output I got when I first ran the program. It's actually very odd as whilst the code guarantees that any increment operation or decrement operation will complete without interference, it doesn't guarantee that all the increment operations will happen before the subtract operations. Indeed if it did it would be negating the usefulness of threads. Subsequent runs varied the orders of the increment and decrement operations, however in all cases each output line consisted of either "Add" or "Subtract" followed by a number which was either one more or one less than the number on the previous line.

Use of the `synchronize` modifier enables the programmer to determine what set of operations are regarded as 'atomic' i.e. cannot be interleaved with operations from another thread.

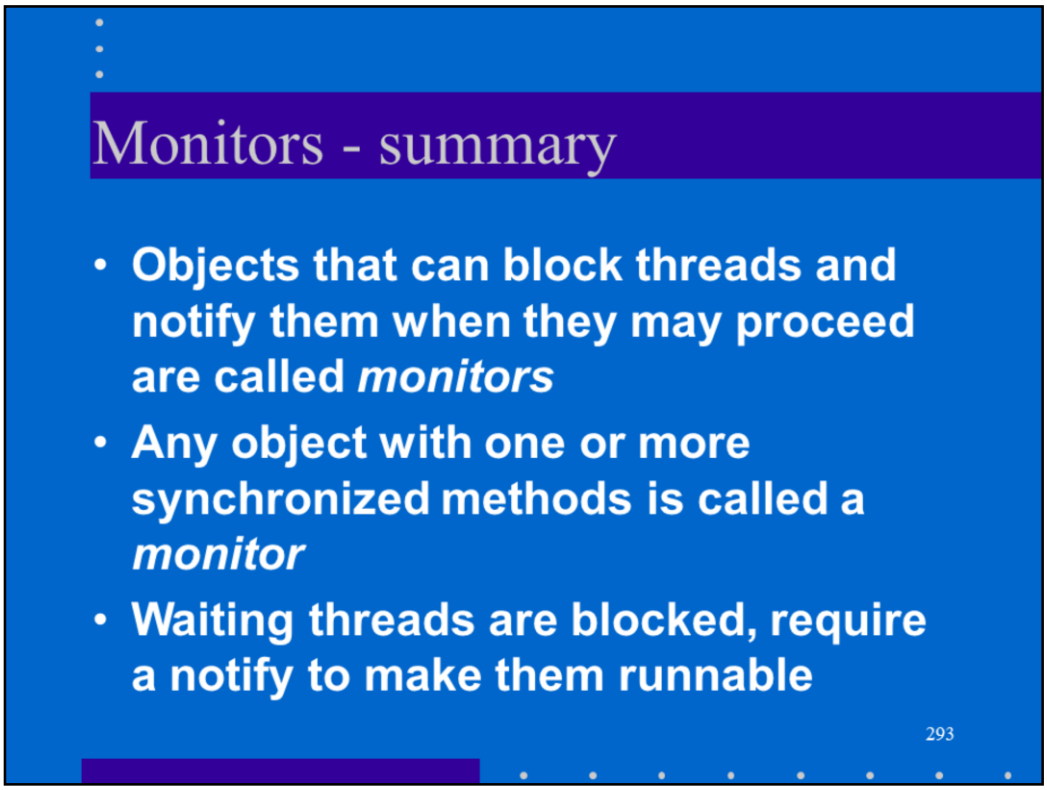
It is possible to apply the `synchronize` modifier to a block as well as to a method.



On another run of the program these results were obtained. Here the threads are running alternately. There is nothing in the code which makes them do this it just happened that way. One was added ten times and was subtracted ten times. Each printed result is one different from the one before.

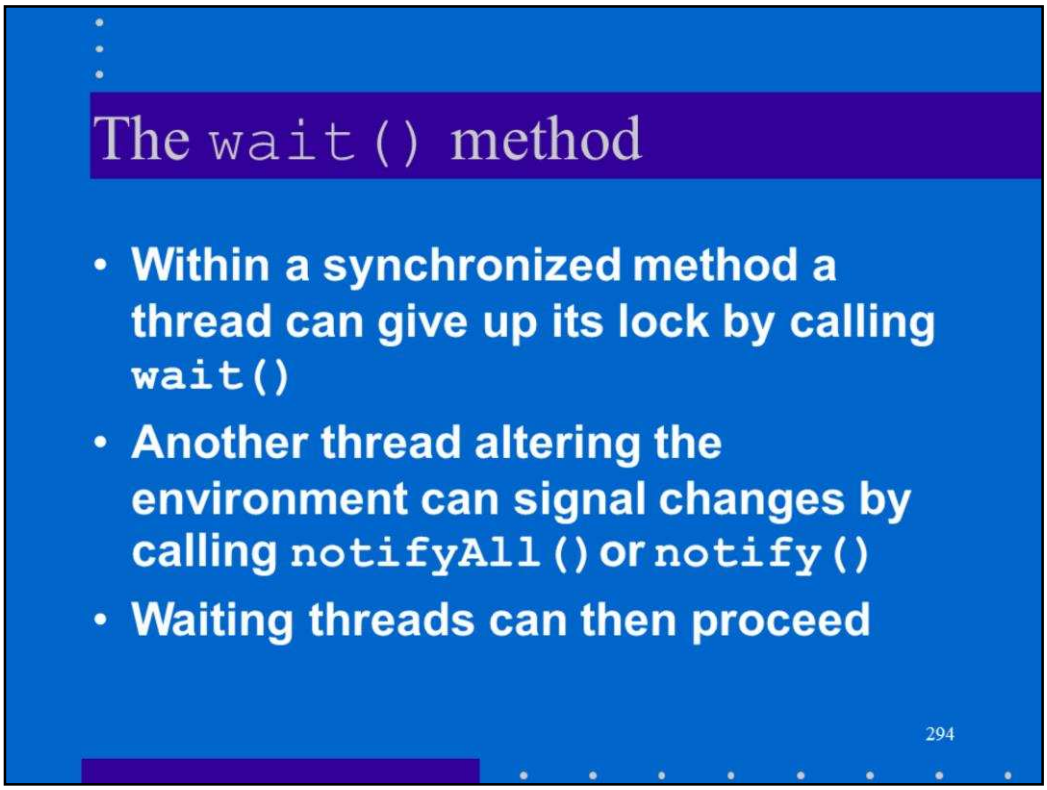


Here the Add thread ran 7 times before Subtract was given an opportunity. But again every result is one more thyan the previous and the final result of 0 is predictable. Each output line consists of either Add or Subtract followed by a number which shows that the operations are now atomic.


Monitors - summary
• Objects that can block threads and notify them when they may proceed are called *monitors*
• Any object with one or more synchronized methods is called a *monitor*
• Waiting threads are blocked, require a notify to make them runnable
293

The term *monitor* is generally understood as that described in Hoare's paper. In Java, a monitor is an object which can make a thread wait and then subsequently let it proceed. Any object which contains a synchronized method is a monitor.

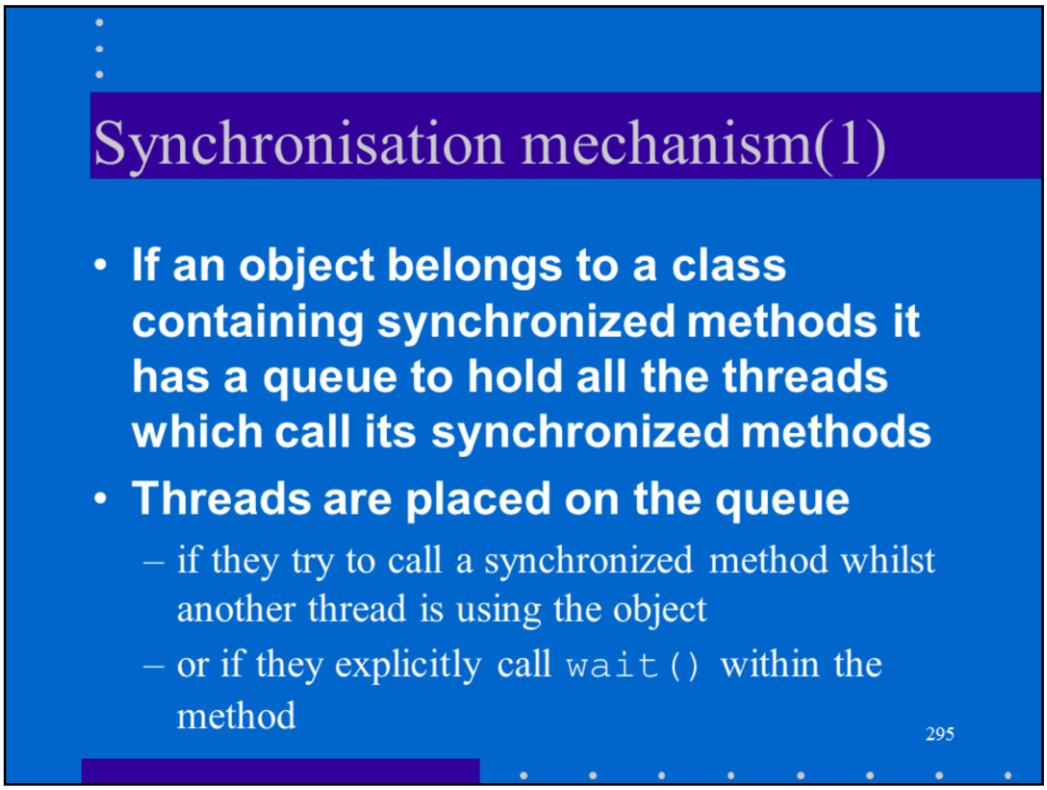
If a thread invokes `wait()` whilst executing code in a monitor it is in the blocked state and is made runnable when a `notify()` operation is carried out on the object that has caused it to wait.


The wait() method
• Within a synchronized method a thread can give up its lock by calling wait()
• Another thread altering the environment can signal changes by calling notifyAll() or notify()
• Waiting threads can then proceed
294

Once a thread has invoked a method it may discover that it cannot proceed. For example, a bank object may permit the transfer of money from one account to another. Transferring an amount X from one Account A to Account B cannot take place if A contains less than X pounds. A thread may enter the synchronized method for transferring money and discover that, due to insufficient funds in the account it cannot proceed. In these circumstances the thread may call `wait()` which will place the thread in a blocked state and allow other threads to invoke this object's synchronized methods.

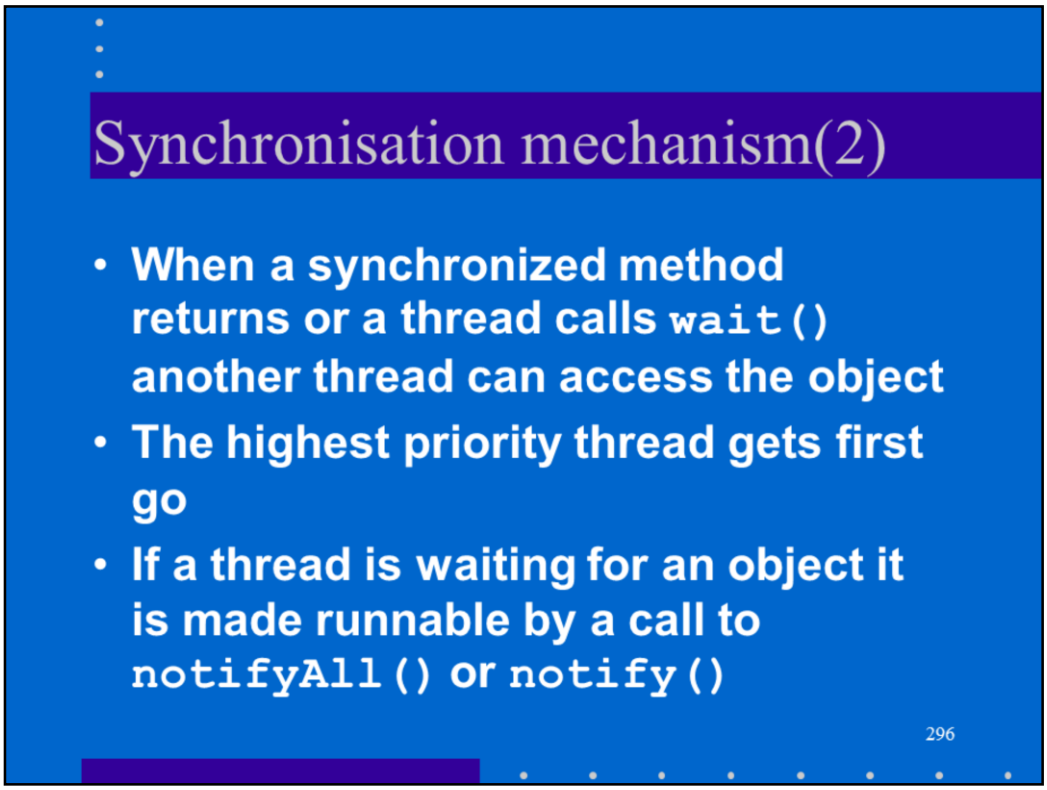
Subsequent threads may transfer money into Account A. Every thread which alters the money in any account calls `notify()` or `notifyall()`. The `notify()` method makes one randomly selected thread, which is waiting for the object, runnable. The `notifyAll()` method makes all threads waiting for the object runnable.

The methods `wait()`, `notify()` and `notifyAll()` are all methods of class `Object` and may only be called from a synchronized method.


Synchronisation mechanism(1)
• If an object belongs to a class containing synchronized methods it has a queue to hold all the threads which call its synchronized methods
• Threads are placed on the queue
– if they try to call a synchronized method whilst another thread is using the object
– or if they explicitly call wait() within the method
295

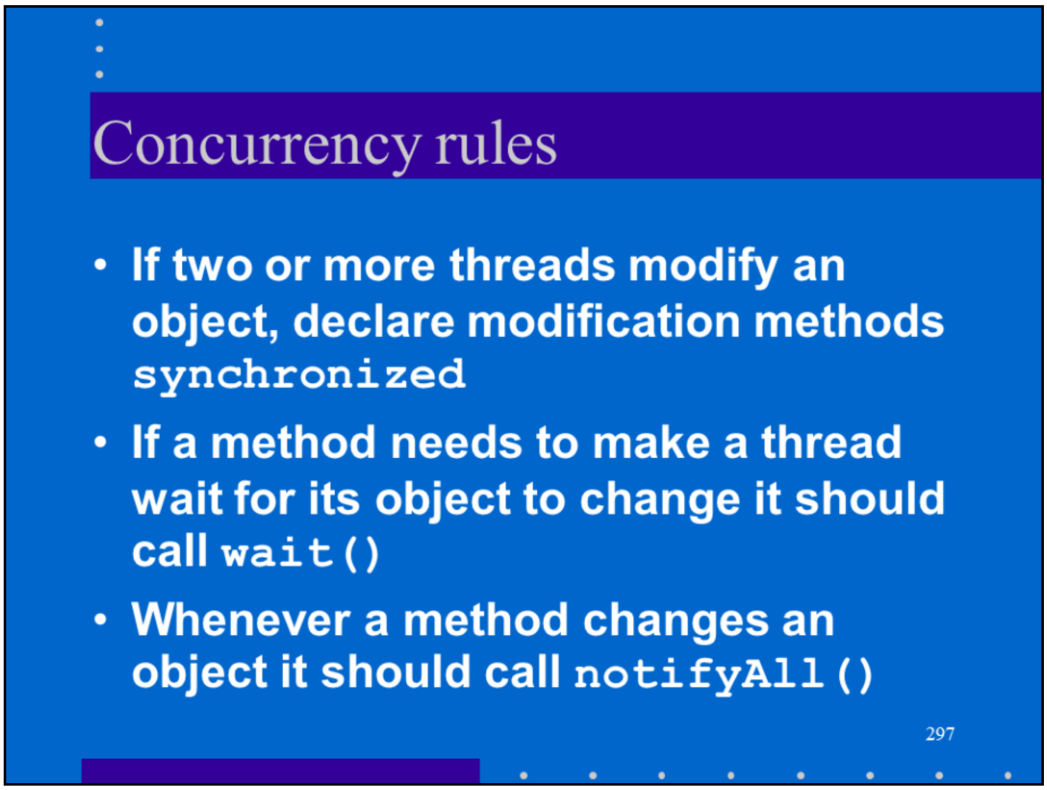
Threads that are waiting to enter a method will begin execution of that method from the first statement of that method when they are permitted to continue.

If a thread has executed a call to `wait()` within a method it will continue to wait until another thread issues a call to `notify()` (and it is the object which is randomly selected to be notified) or `notifyAll`(). When the thread ceases to wait it will begin execution at the statement which follows the statement which invokes `wait()`.


Synchronisation mechanism(2)
• When a synchronized method returns or a thread calls wait() another thread can access the object
• The highest priority thread gets first go
• If a thread is waiting for an object it is made runnable by a call to notifyAll() or notify()
296

A call to wait() must always appear in a `try…catch` block since waiting threads can be interrupted and throw an `InterruptedException`.

If there are multiple invocations of `wait()` in the methods for an object it is generally more sensible to use `notifyAll()` than `notify()` as `notify()` only restarts one randomly chosen thread and it could be that this thread still cannot proceed. Threads which are notified should therefore check to see if the condition that caused them to wait still applies. For this reason, invocations of `wait()` often appear in a `while` loop. The loop terminates when the waiting condition is no longer true.


Concurrency rules
• If two or more threads modify an object, declare modification methods synchronized
• If a method needs to make a thread wait for its object to change it should call wait()
• Whenever a method changes an object it should call notifyAll()
297

When two threads operate on the same object the results can often be unpredictable. To prevent this, methods which modify objects should be written so that when taken as a whole the modifications they perform leave the object in a consistent state. These modification methods should be declared as `synchronized` so that only one thread at a time can execute them.

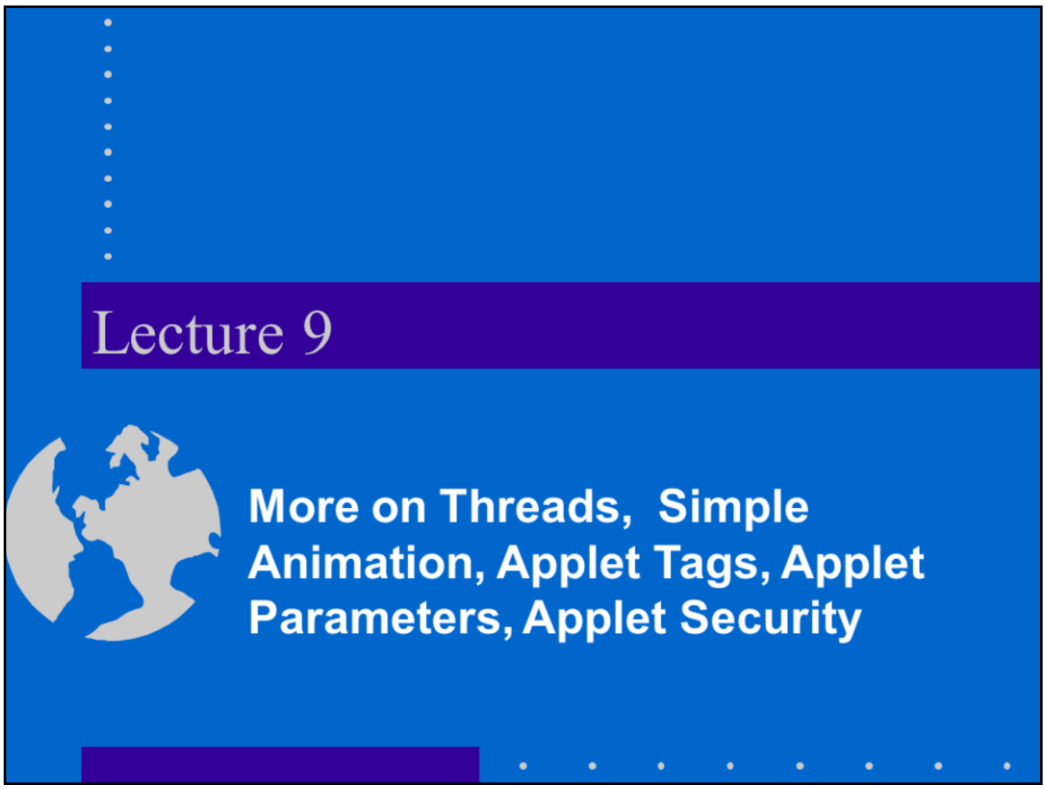
There will be certain situations where threads cannot proceed because the conditions necessary for them to continue are not currently met. For example, a print spooler cannot print anything if the print queue is empty. In these circumstances the method can invoke `wait()` which will halt the thread until a method in a different thread which adds to the print queue calls `notify()` or `notifyAll()`.

Synchronized methods that make use of `wait()` should call `notifyAll()` before they exit to free up waiting threads. Methods that call `wait()` should ensure that when they are notified they check whether the condition that caused them to wait in the first place still applies before proceeding. If this condition still applies, they should call `wait()` again.

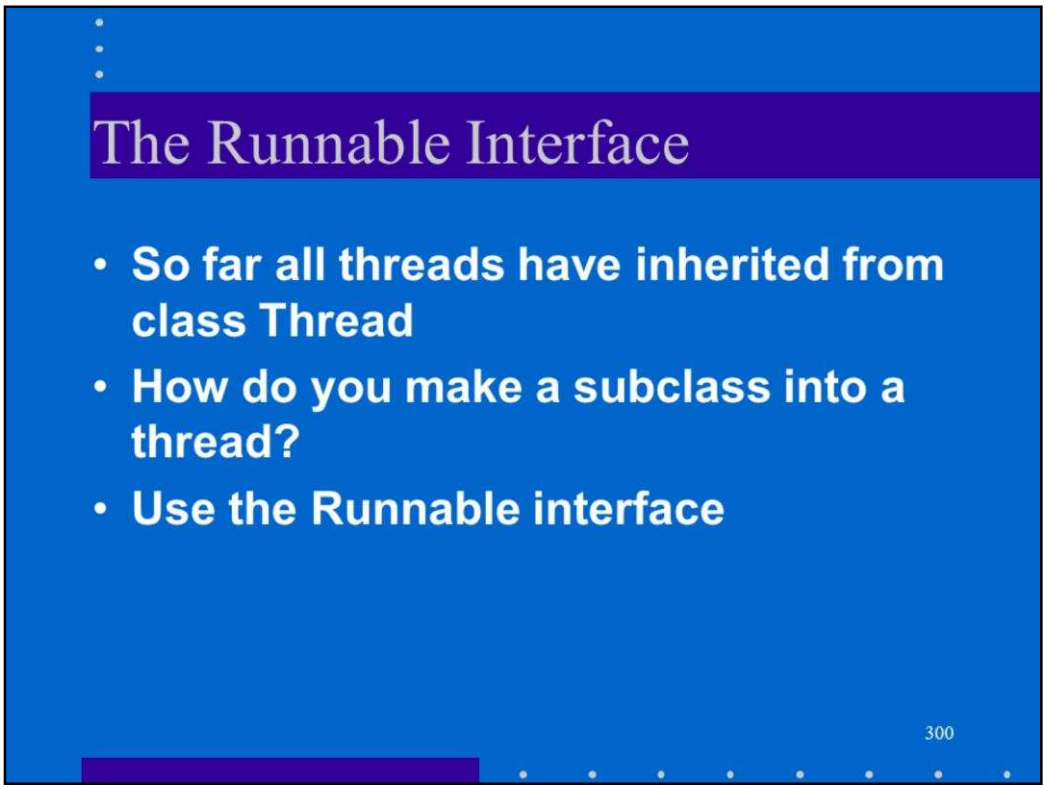
Even with these guidelines, programming concurrent threads is still tricky and careful thought will be required to ensure that the results are what was intended.

Summary

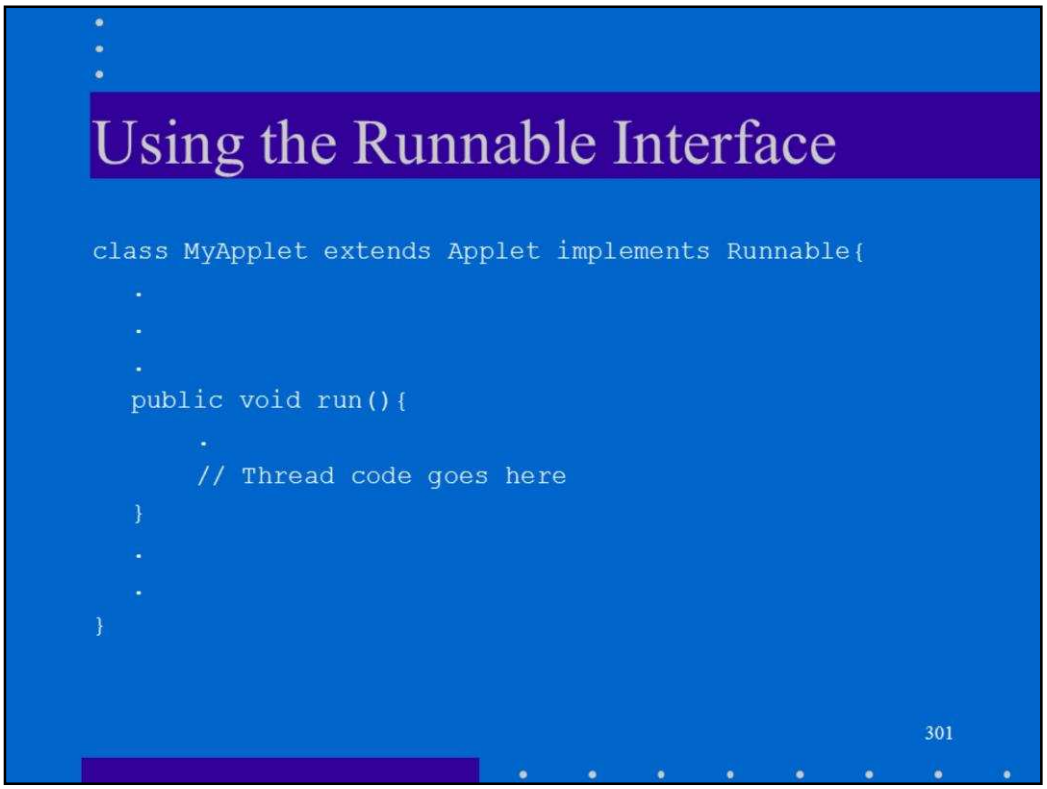
- Multitasking
- Threads
- Multithreading
- The `Thread` class
- Thread states
- Priorities
- Selfish Threads
- Synchronising threads
- Monitors
- `wait()`
- `notifyAll()`



This lecture completes the material on threads begun in lecture 8. An example of using a thread to create a simple animation is presented in some detail. Also included is a look at the hooks within HTML (Hyper Text Mark-up Language) which are used to embed an applet in a web page. Finally there is a discussion of the way Java ensures that applets do not act like viruses.


The Runnable Interface
• So far all threads have inherited from class Thread
• How do you make a subclass into a thread?
• Use the Runnable interface
300

In lecture eight every example of a thread was implemented by using a class which extended class `Thread`. This will not always be possible. For example, every applet must inherit from class `Applet`. But Java only supports single inheritance and therefore if something is a subclass of Applet it cannot be a subclass of `Thread`. Here is a place where we might wish to use multiple inheritance. An earlier lecture asserted that interfaces were Java's mechanism for simulating multiple inheritance. In this case the `Runnable` interface can be used to make a class, which already extends another class, take on properties which will enable it to behave as though it were a subclass of `Thread`.



The code that must be supplied is fairly simple. Here we have an applet which, as with all other applets, extends class Applet. It also implements the `Runnable` interface. The definition of the `Runnable` interface is:

```
public interface Runnable {
          public abstract void run();
}
```

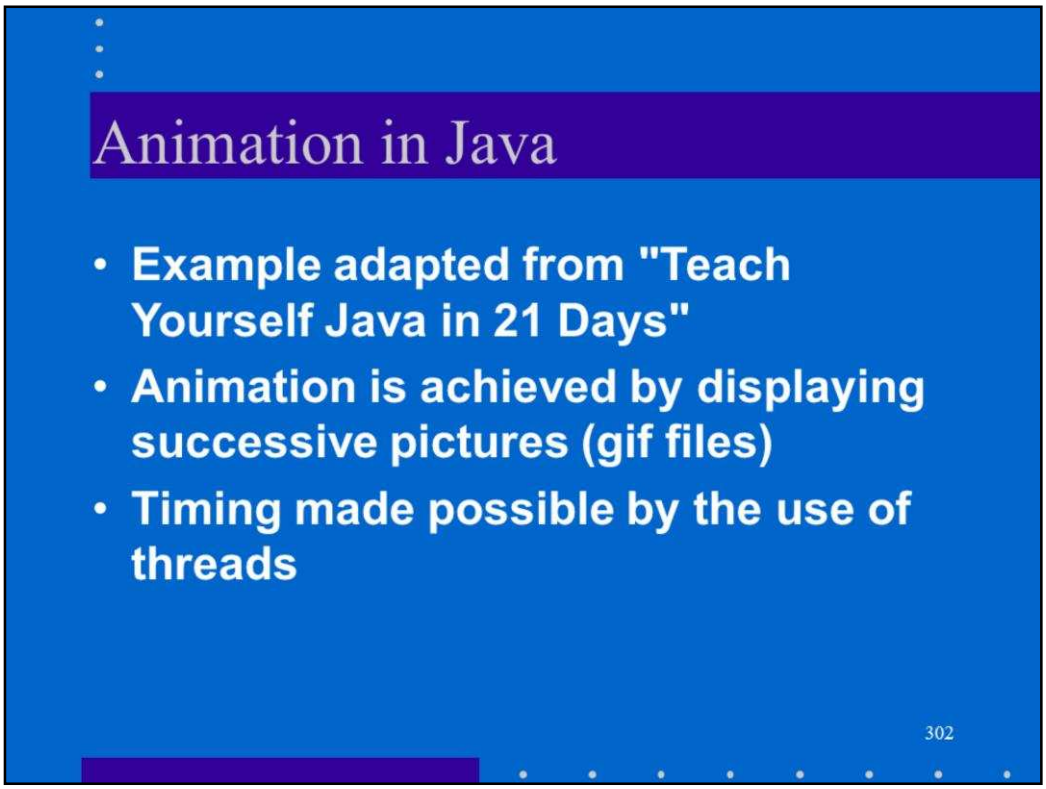
In other words any class which implements `Runnable` must have a `run()` method. This will contain the code which will determine what the thread does when it is executed.

Class `Thread` contains a constructor which will take an object from a class which implements `Runnable` and create a instance of `Thread` from it.

The following piece of Java could be used to execute the code in the `run()` method of `MyApplet` in its own thread:

```
Thread myThread = new Thread(this);
myThread.start();
```

An object of class `Thread` is created using the constructor which takes an object of type `Runnable` (in this case `this`). This object will respond to all the messages supported by the `Thread` class.

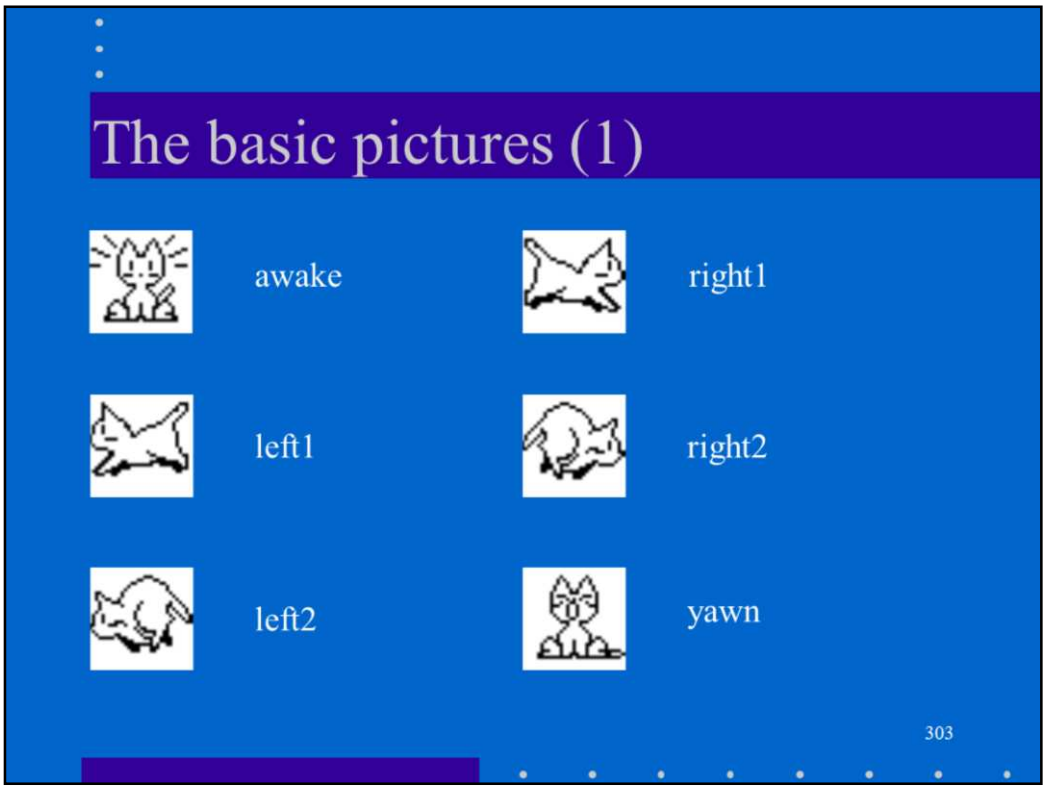

Animation in Java
• Example adapted from "Teach Yourself Java in 21 Days"
• Animation is achieved by displaying successive pictures (gif files)
• Timing made possible by the use of threads
302

The following slides show how a very simple animation can be constructed using Java. A number of very sophisticated features for aiding programmers to produce animation effects are available in the Java class library. For simplicity's sake none of them have been used. The aim of the program is not to demonstrate how animations should be coded but rather to bring together a number of the features of Java that have been discussed in this and previous lectures. In particular, it shows the use of events and threads.

The example originally appeared in the book "Teach Yourself Java 1.2 in 21 Days", Lemay, L. and Cadenhead, R, SAMS, ISBN: 1-57521-390-7 however it has been considerably extended from the version contained therein.

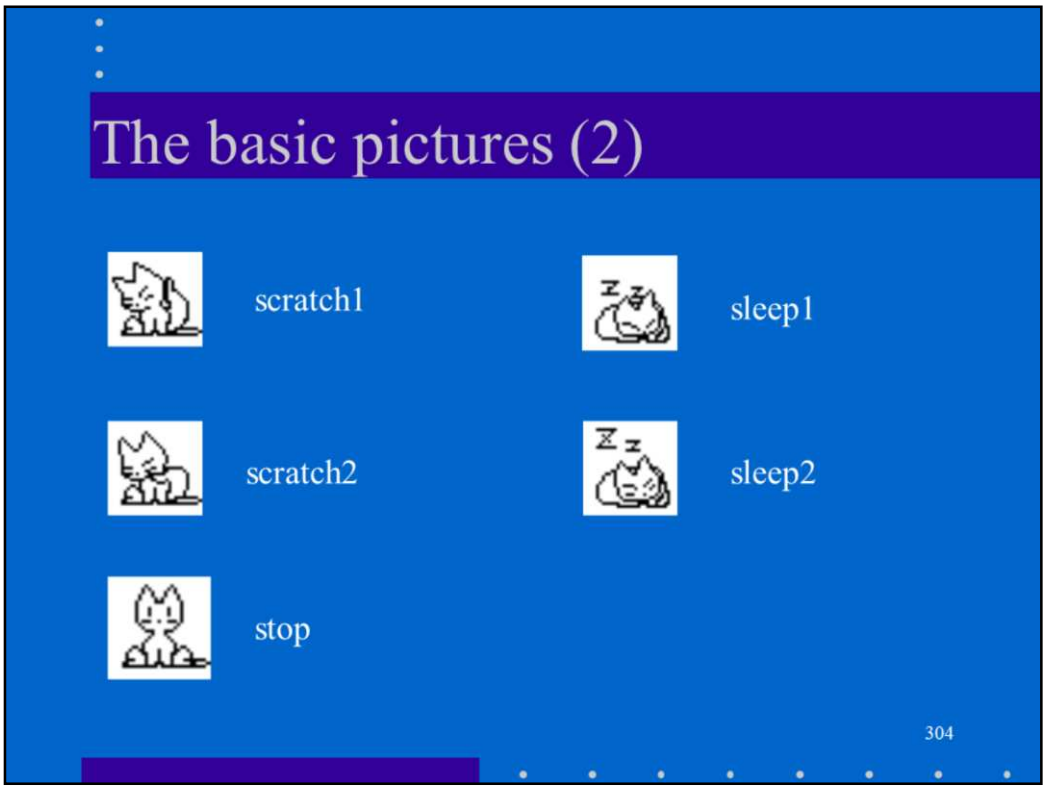
Animation is achieved by the successive display (and movement) of a set of gif files (remember that Java can load images from jpeg or gif files).

The effect is rather like that of a flip page book where each page contains an image slightly different from the page before and movement is simulated by flipping the pages of the book. This technique only works if the speed at which each successive page is revealed can be controlled. In this example, the animation speed is controlled by using the `sleep()` method of the `Thread` class.

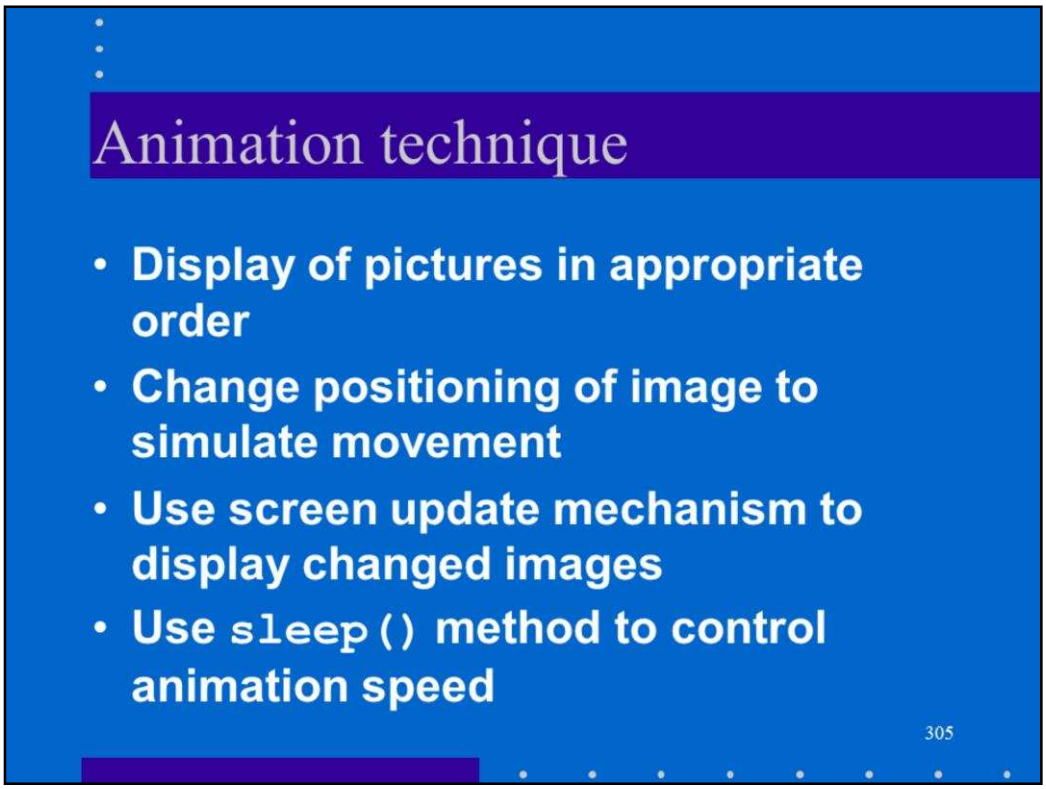


There are eleven pictures used to create the animation effect. The cat has only a few basic movements; it can run to the left; it can run to the right; it can sleep; it can yawn; it can scratch.

Running to the left is simulated by displaying the image left1 followed by left2 followed by left1 and so on. Each time the image is displayed it is moved a little more to the left within the display window. Running to the right works in the same way except that this time the image moves to the right and the images right1 and right2 are used.



Scratching is simulated by alternately displaying scratch1 and scratch2. In this case the image is not moved within the window. Sleeping is simulated in the same way as scratching, except that sleep1 and sleep2 are used. The images awake, yawn and stop act as intermediate pictures between the main action sequences.

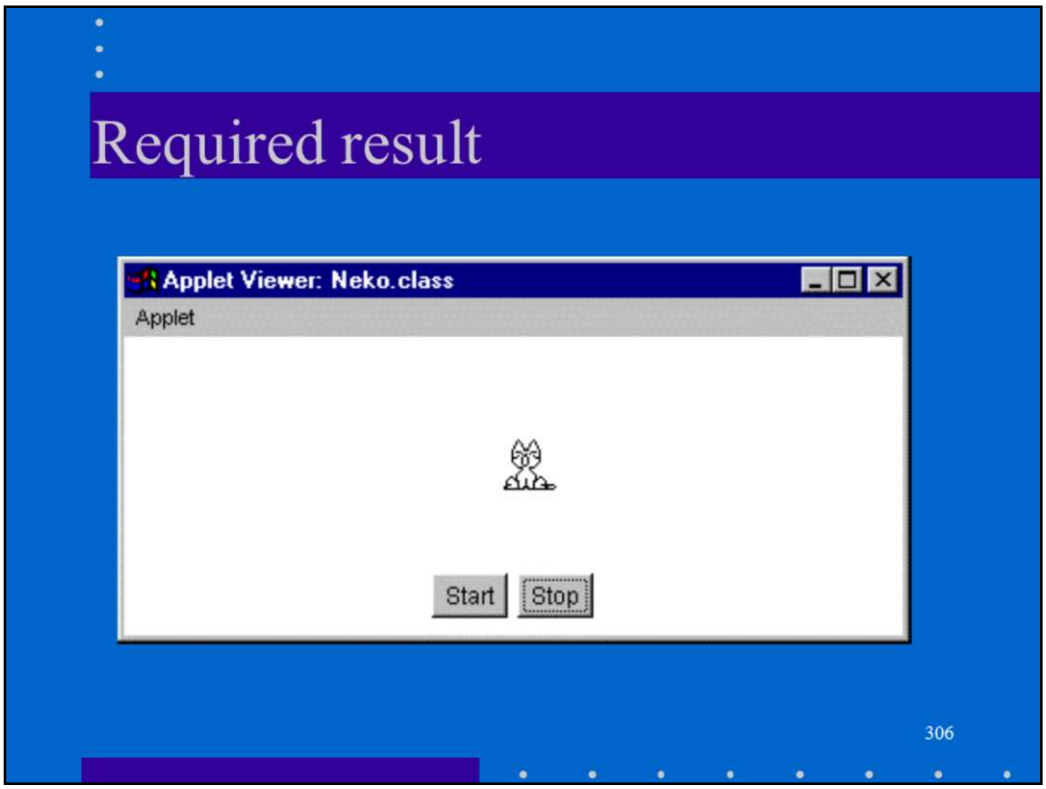

Animation technique
• Display of pictures in appropriate order
• Change positioning of image to simulate movement
• Use screen update mechanism to display changed images
• Use sleep() method to control animation speed
305

To make the animation work the program has to display the pictures in the appropriate order. This is fairly easy to manage.

Slightly more complicated, although not too difficult, is the problem of controlling the movement of the pictures which represent running across the window. One major issue is turning the cat round when the edge of the window is reached.

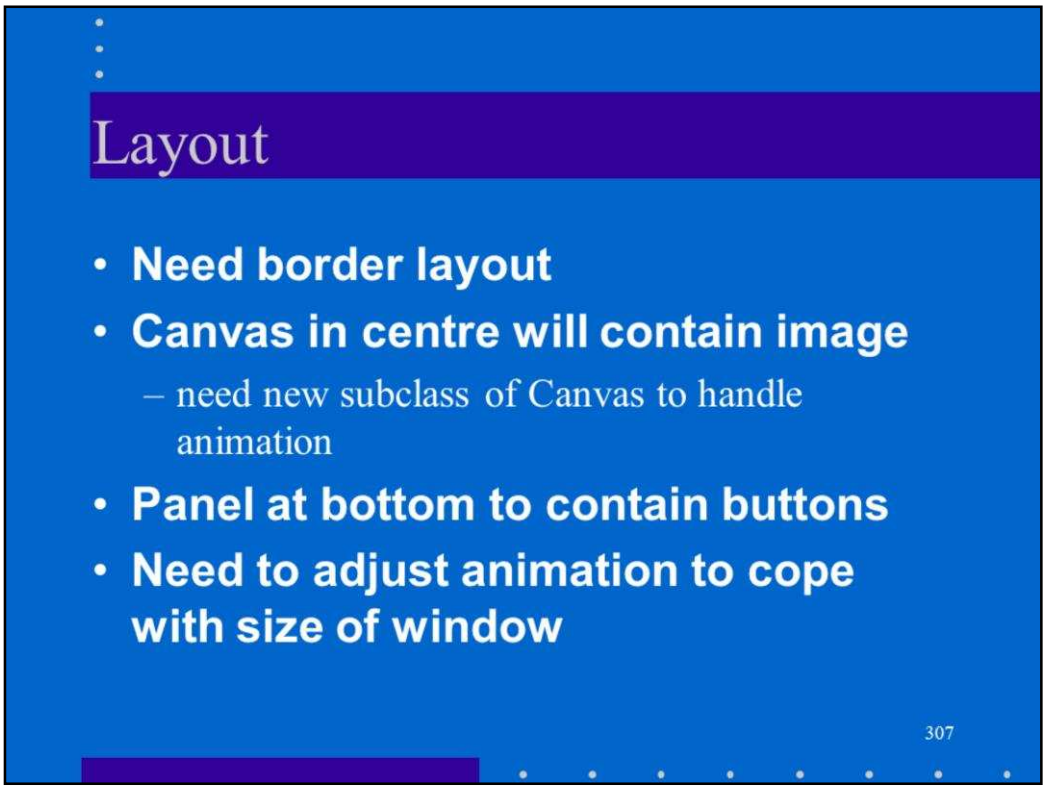
Switching between images can be achieved via the Java screen update mechanism. Whenever the Java virtual machine decides that the screen needs updating it will call the `update()` method of a component. This will erase the existing contents of the component and then call its `paint()` method. Redrawing successive images is simply a matter of making the results of calling `paint()` vary, and scheduling screen updates via calls to the `repaint()` method.

The animation won't work if the images are switched too quickly. The technique relies on the fact that an image will be displayed long enough for someone to notice it. Therefore there is a need to have some method of controlling the gap between when a picture is displayed and when it is erased. The `sleep()` method of class `Thread` provides a way in which a thread can idle for a given period of time.



This slide shows the results we want to achieve in this applet. The cat will initially appear at the left of the window. It will run right to the centre of the window, then yawn, then scratch, then sleep, then wake and run off again to the right. When it reaches the right hand side of the window it will turn round and run to the left. It will stop again to sleep in the middle and then wake up and run left until it reaches the left hand side of the window when it will turn around and begin again.

When the animation is running, pressing the button labelled "Stop" will freeze it. Pressing the button labelled "Start" will restart the animation if it is frozen. Stopping and starting the animation can take place wherever it is in the cycle of activities and not just at predefined points.

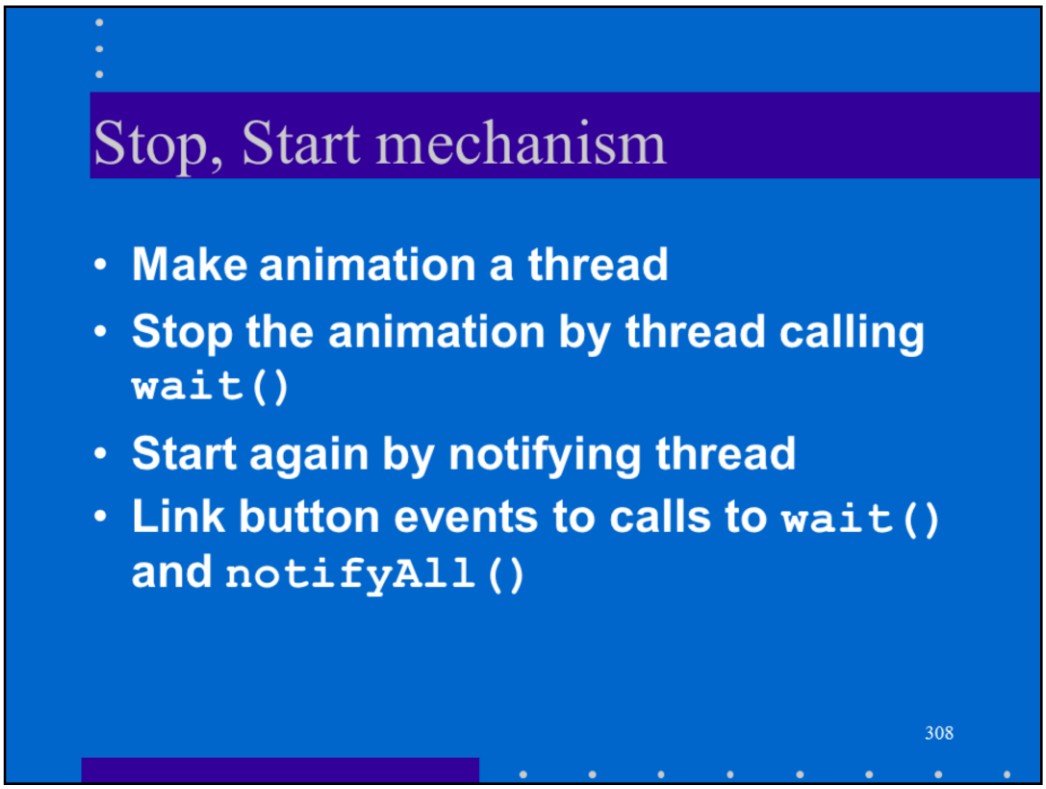

Layout
• Need border layout
• Canvas in centre will contain image
– need new subclass of Canvas to handle animation
• Panel at bottom to contain buttons
• Need to adjust animation to cope with size of window
307

It is clear that neither flow layout or grid layout can achieve this type of display. A border layout is necessary. We only need to use two of the display positions though, `south` and `center`. In one position we will need to place an object of class `Canvas` to display the graphics in the other position an object of class `Panel` which will contain the buttons.

A canvas in which we can override `paint()` will be used in the center position.

Since we wish to place two buttons in the south position, and normally a border layout only allows a single component in each of its five positions, we need to use a panel for the buttons. This panel will use flow layout for positioning the buttons.

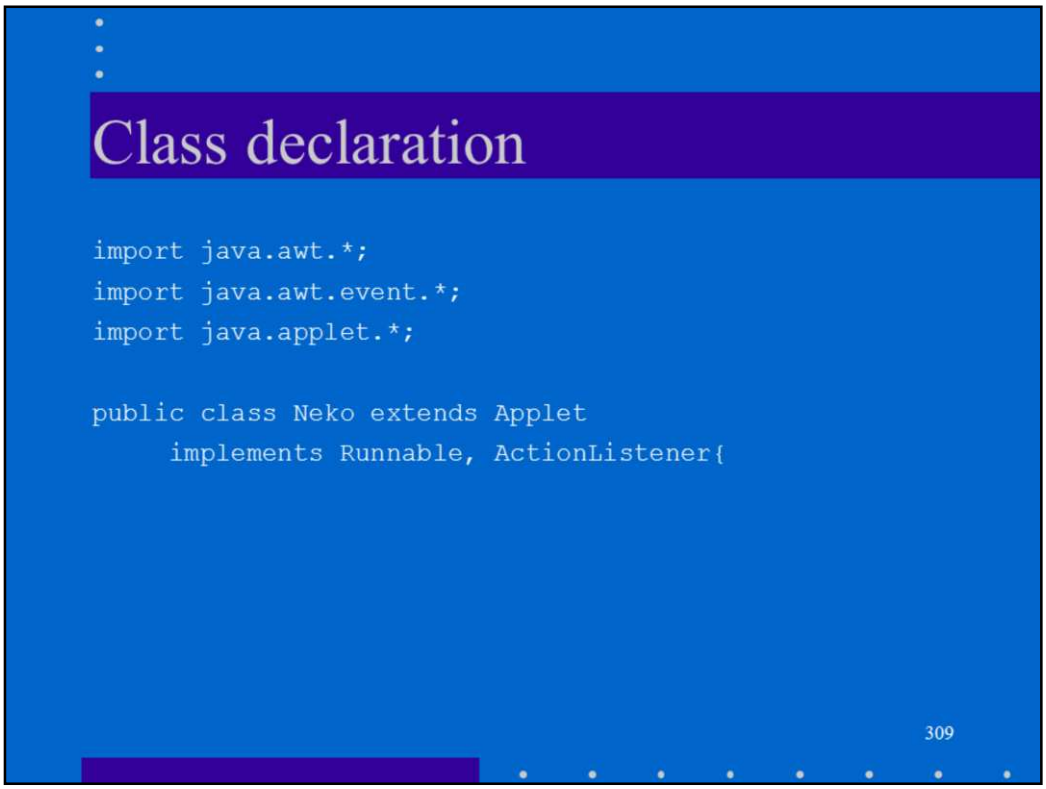
Obviously, at any time the user could decide to resize the window. It is required that the cat always runs to the edge of the window. The pixel positions at which it turns round will therefore be affected by the current window size. The current window size for the applet can be found by calling the method `getSize()`.


Stop, Start mechanism
• Make animation a thread
• Stop the animation by thread calling wait()
• Start again by notifying thread
• Link button events to calls to wait() and notifyAll()
308

A feature of the animation is that we want to be able to stop and restart it by pressing the buttons. The button presses will generate events, and we can perform the actions necessary to start or stop the animation within an event handler. But how can we stop and restart the action?

If the animation takes place in a thread the mechanism for doing this is simple. Any running thread can call `wait()` and any waiting thread can be notified. Class `Thread` implements the methods `wait()` and `notifyALL()` which allows this.

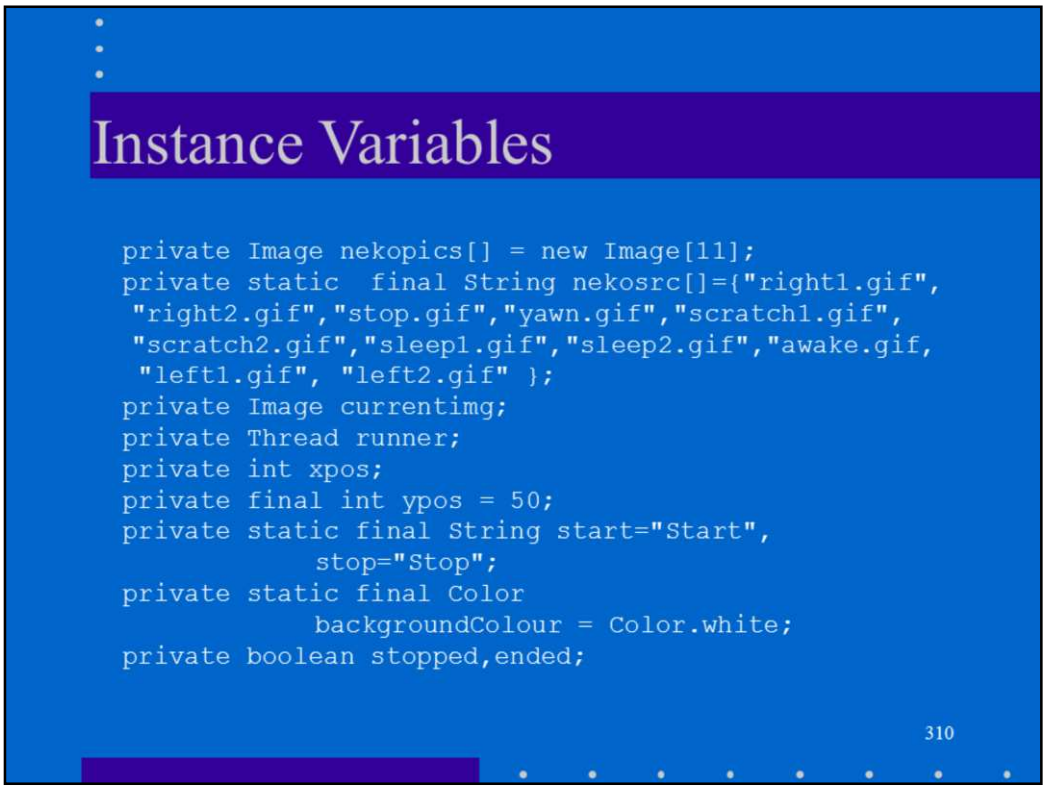
The animation is therefore given its own thread. To achieve this, and make the program into an applet the class must extend class `Applet` and implement interface `Runnable`.



This slide and the following slides show the code of the entire applet.

The applet will manipulate graphics so it needs to import package `java.awt`. The buttons will generate events so it also needs `java.awt.event`. As it is an applet it needs `java.applet`.

The public class is called `Neko` and, as with all applets, it extends class `Applet`. Since it will carry out the animation in a thread it has to implement interface `Runnable`. This means that a method `run()` must appear in the definition of the class. The applet will also deal with events generated from button presses. Consequently, it must also implement interface `ActionListener`. This will mean that it must contain a method `actionPerformed()`, which will contain code to handle the button events.



These are the instance variables declared for the class. All of them are `private` because there is no need to access them from outside the class

`nekopics` is an array of images. The individual pictures are read from the files they are stored in and placed in this array.

The names of the files where the pictures are stored appear in `nekosrc` which is an array of strings. This will be constant and therefore is declared as a `final` variable.

The variable `currentimg` is used to reference the image which is currently on display. This variable is used to communicate what should be drawn.

`runner` will hold a reference to the thread in which the animation will run.

`xpos` is a pixel co-ordinate reference to the current position of the image.

The image will move from side to side, not up and down. Its vertical position can therefore be represented by a constant value `ypos` in this case set to 50. The image is always drawn at pixel `xpos,ypos`.

`start` and `stop` are string constants which define labels for the buttons.

`backgroundColour` is another constant used to set the background colour of the applet window. White is a sensible choice here as the background colour in the gif files is white and a white background will hide this.

`stopped` and `ended` are boolean flags used to indicate when the animation should stop and when the thread should terminate.

Canvas subclass

```
class AnimationCanvas extends Canvas{
  public AnimationCanvas(){
    setBackground(backgroundColour);
  }

  public void paint(Graphics g){
    if(currentimg != null)
        g.drawImage(currentimg, xpos, ypos,this);
  }
}

private AnimationCanvas ac= new AnimationCanvas();
```

311

`AnimationCanvas` is an inner class which is used to create the object placed in the centre of the screen (i.e. the animation). It is a subclass of `Canvas` because it needs to be a canvas with an overridden `paint()` method.

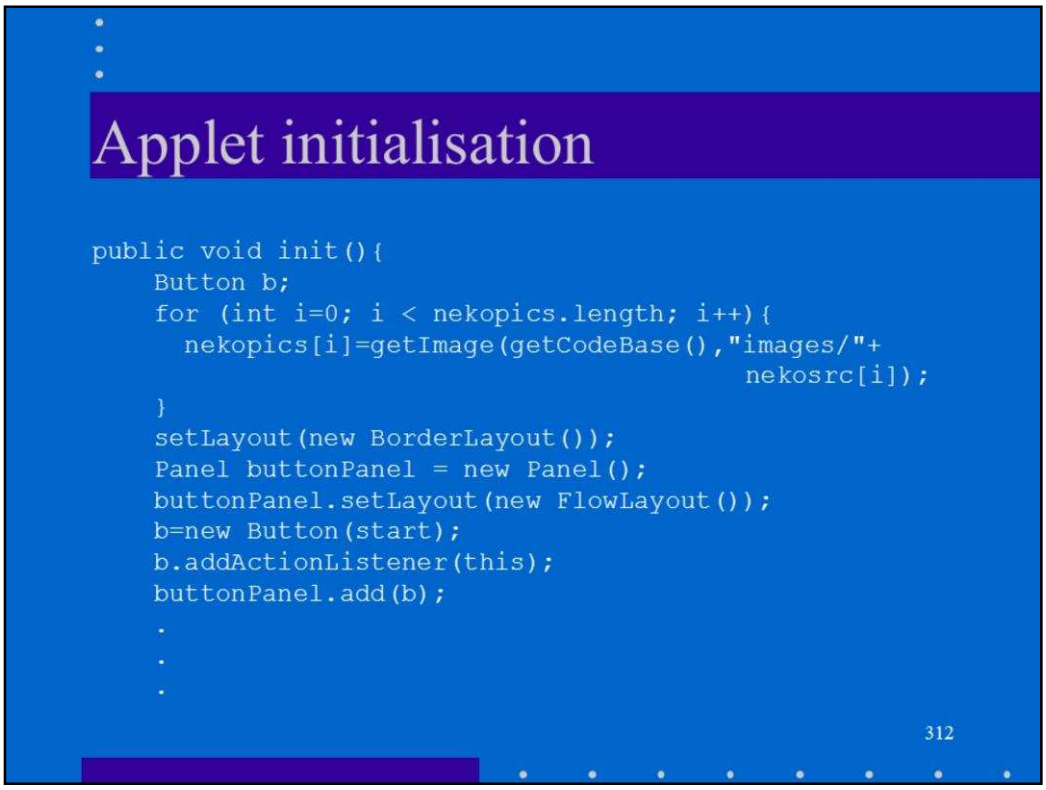
An inner class is useful here because it is helpful to have access to `currentimg`, `xpos` and `ypos`.

The constructor sets the background colour of the panel.

The `paint()` method draws the image referenced by `currentimg` at pixel position `xpos, ypos`. This method will be invoked by the Java runtime environment whenever the Screen Updater decides to update this screen component.

There is a test to check that `currentimg` has a value because it is possible for `paint()` to be invoked before the `currentimg` gets a value assigned to it.

`ac` is an instance variable belonging to class `Neko` and is initialised to reference an `AnimationCanvas` object.



Every applet may override an `init()` method. `init()` methods normally contain initialisation code for an applet.

This `init()` method loads all the gif files, from a subdirectory called image, into images in the `nekopics` array using a `for` loop. `getImage()` is a method of class `Applet`:

`public Image getImage(URL url, String name)`

Returns an `Image` object that can then be painted on the screen. The url argument must specify an absolute `URL`. The name argument is specified relative to the url argument.

`getCodeBase()` is another applet method:
`public getCodeBase()`
Gets the base `URL`. This is the `URL` of the applet itself .

The effect of this is that the images are fetched from a subdirectory *images* of the directory where the applet code is stored.

The layout of the applet is set to border layout.

A panel is created to hold the buttons. This is given a flow layout. A button is created using the string contained in the variable `start`. The applet is nominated as the event handler for this button. The button is added to the panel.

Applet initialisation continued

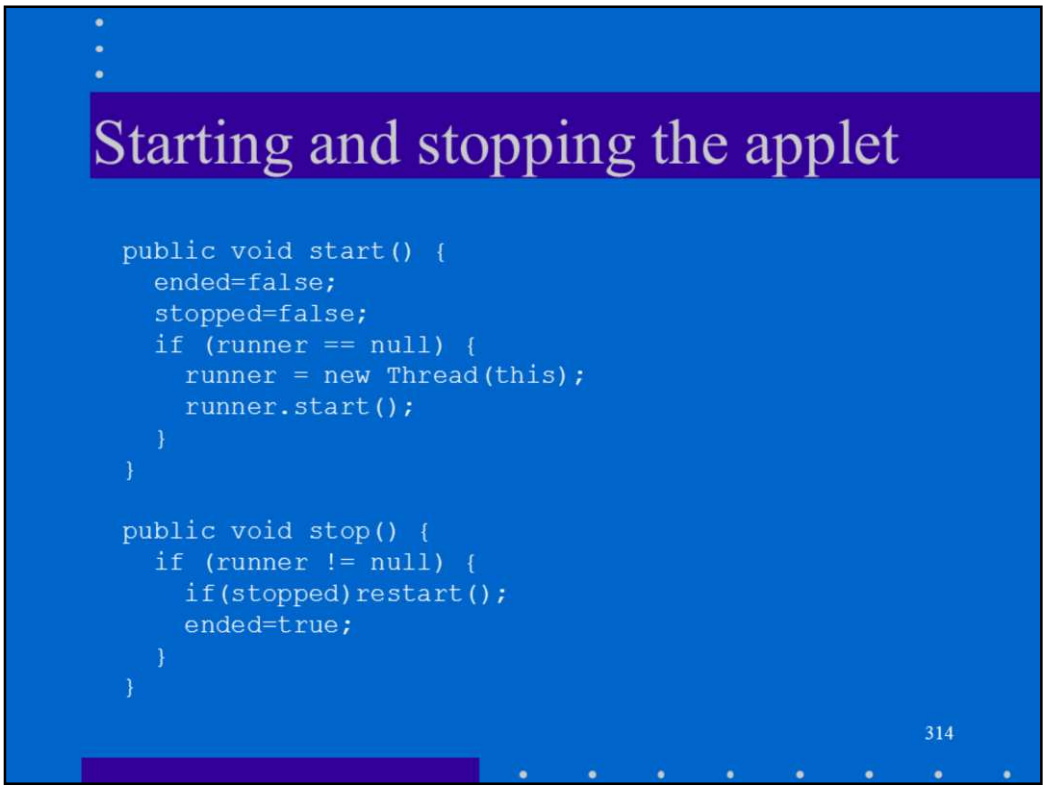
```
    .
    .
    .
    b=new Button(stop);
    b.addActionListener(this);
    buttonPanel.add(b);
    add("Center",ac);
    add("South",buttonPanel);
}
```

313

This code continues the `init()` method.

A button is created using the string referenced by the `stop` variable. The applet is nominated as the event handler for this button. The button is added to the button panel. As the button panel uses flow layout, the position of this button will follow on from the position of the previous button.

The animation canvas is placed in the centre of the applet's window. The button panel is placed at the bottom of the applet's window.



Applets may override the methods `start()` and `stop()`.

`start()` is normally invoked by a web browser to tell the applet to begin executing.

`stop()` is called by the browser when a new web page is loaded or when the window is iconised and the applet should stop executing.

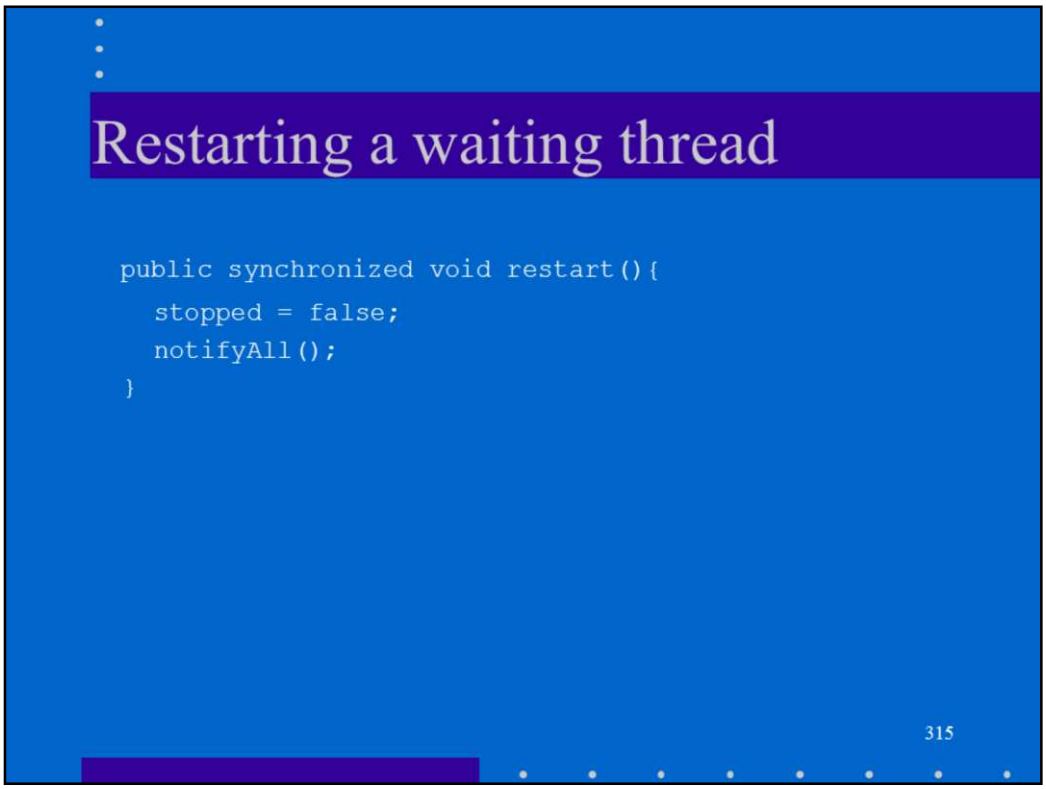
The `start()` method creates a new thread if one does not currently exist and puts it into the runnable state. `runner`, a instance variable of the applet, is used to refer to the thread. This is assigned a thread created by passing a reference to the applet object (`this`) to the `Thread` constructor. The code is legal because the Neko applet implements the `Runnable` interface. When the `start()` method of class `Thread` is invoked for the object referenced by `runner`, the thread is passed to the Java Virtual Machine for scheduling .

It is important to realise that there are two different `start()` methods in this code. Both threads and applets have `start()` and `stop()` methods. In order to make sense of the code it is important to establish whether the message call is sent to an object of class `Applet` or an object of class `Thread`.

Applying the `start()` method to runner which references an object of class `Thread` effectively causes the code in the `run()` method of class `Neko` to execute in a separate thread.

The `stop()` method of the applet causes the termination of the thread

referenced by `runner` by setting the `ended` flag this will eventually cause the thread run method to exit.

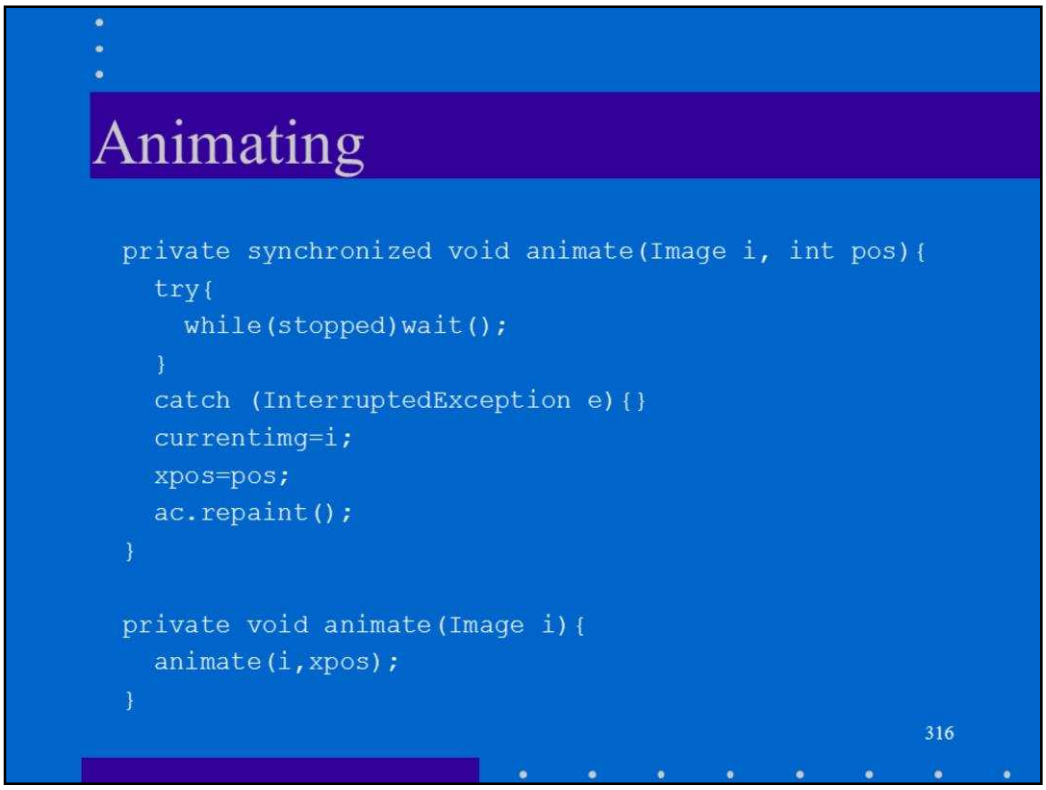


This method is called whenever the thread may need restarting. We shall see that it is called later on by the code that deals with the start button.

All it does is to reset the stopped flag and notify all waiting threads. Any threads which cannot proceed when the stopped flag is false will now be able to continue.

The method must be declared as `synchronized` in order for it to call `notifyAll()`.

The `stop()` method of the applet also calls `restart()`, it does this because it is impossible for the thread to terminate while its is blocked by a `wait()` statement. `stop()` restarts the thread after setting a flag which will cause it to terminate at some later point.



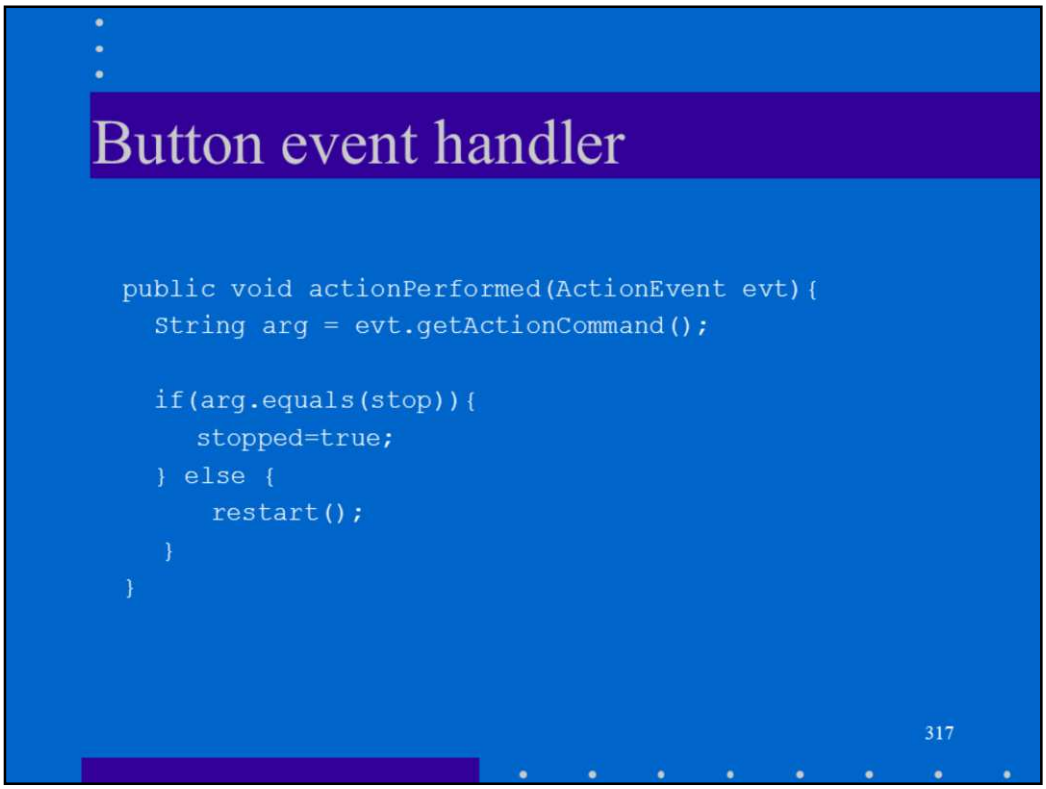
Animation is achieved by altering the current image and sometimes altering the current position. The variables `currentimg` and `xpos` are changed and then a request is made to the Screen Updater for a redraw (`repaint()`).

Before any animation takes place the `stopped` variable is tested to see if the user has pressed the stop button. If this is the case then the thread makes a call to `wait()` and goes into a blocked state. This has the effect of making the cat stop. When it is notified it retests `stopped` (via the `while` loop) and continues if `stopped` is not set. The method has to be declared with the modifier `synchronized` in order to call `wait()`.

Since `wait()` may throw an `InterruptedException` a `try…catch` block is also needed.

If the `stopped` flag is not set the current image and current position are given new values and a repaint is scheduled for the `AnimationCanvas` object. Eventually the `paint()` method of `AnimationCanvas` will be called and the cat will change image and move.

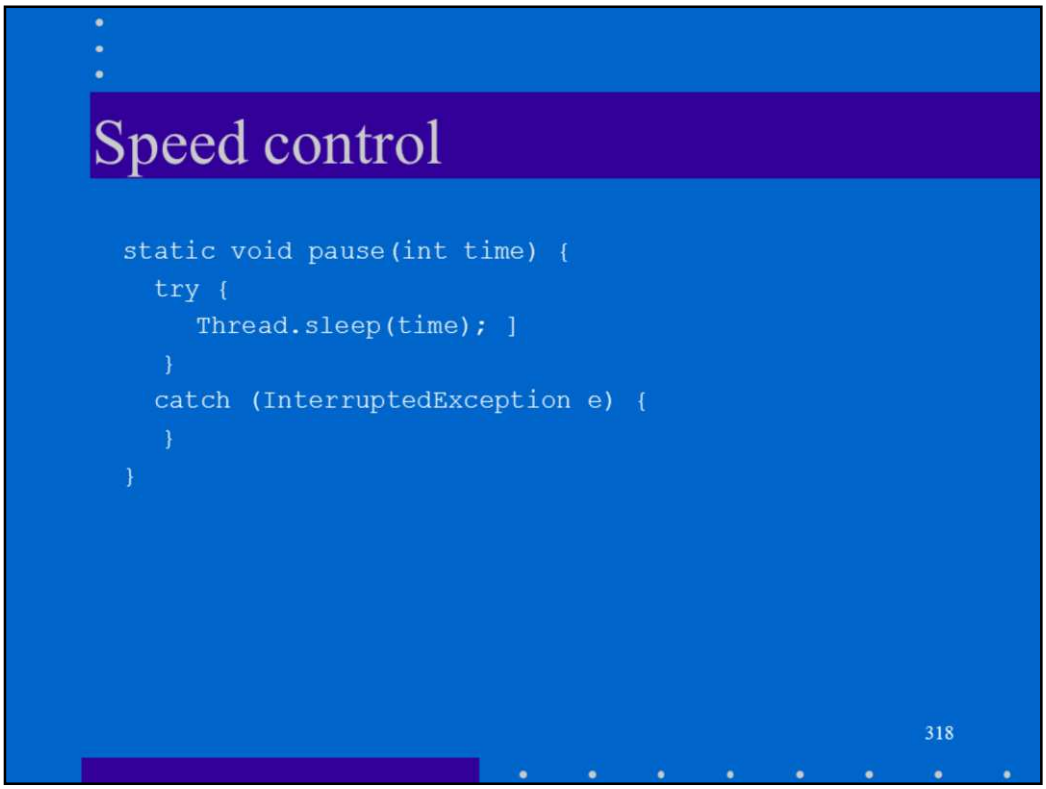
The second version of `animate()` is used for situations where the image changes but remains in the same position. It calls the first version with the current position as a parameter.



The applet was nominated as the event handler for both buttons. When either button is pressed this method will be invoked. The code uses the `getActionCommand()` method of class `ActionEvent` to determine the value of the label on the button that was pressed.

If the stop button is pressed, the `stopped` flag is set to true. Eventually this will cause the `animate()` method to put the thread into a blocked state via a call to `wait()`.

If the start button has been pressed then all threads are notified via a call to `restart()`.



It is necessary in displaying the animation to keep an image on the screen long enough for the viewer to see it. The method `pause()` can be used to do this. All it does is invoke the `sleep()` method of class `Thread`.

There is an exception which can occur when a thread is sleeping. The `pause()` method will catch an exception of class `InterruptedException` and ignore it.

Run to the right

```
void nekorightrun(int start, int end) {
  for (int i=start;i<end;i+=10) {
    // swap images
    if (currentimg == nekopics[0])
         animate(nekopics[1],i);
    else
         animate(nekopics[0],i);
    pause(150);
   }
 }
```

319

This method makes the image appear to run to the right. It alternates the display of the two right running images.

If this method is called when the current image is not the cat running to the right, then the current image is changed to one of the two images showing the cat running to the right.

If the current image is one of the two images of the cat running to the right then the current image is switched to the other image. The parameters of the method are a start pixel co-ordinate and an end pixel co-ordinate. `xpos` (an instance variable of the applet) is successively incremented by ten as each image is drawn until it is beyond the specified end point.

The animation effect is achieved by calling the `animate()` method. After `xpos` and `currentimg` are set up the thread is paused briefly.

Run to the left

```
void nekoleftrun(int start, int end) {
  for (int i=start;i>end;i-=10){
     //swap images
     if (currentimg == nekopics[9])
        animate(nekopics[10],i);
     else
        animate(nekopics[9],i);
     pause(150);
   }
  }
```

320

The same technique is used to make the cat run to the left although this time `xpos` is decremented so that the picture moves towards the left hand side of the window.

Scratch

```
void nekoscratch(int numtimes) {
    for (int i = numtimes; i > 0; i--) {
      animate(nekopics[4]);
      pause(150);
      animate(nekopics[5]);
      pause(150);
    }
  }
```

321

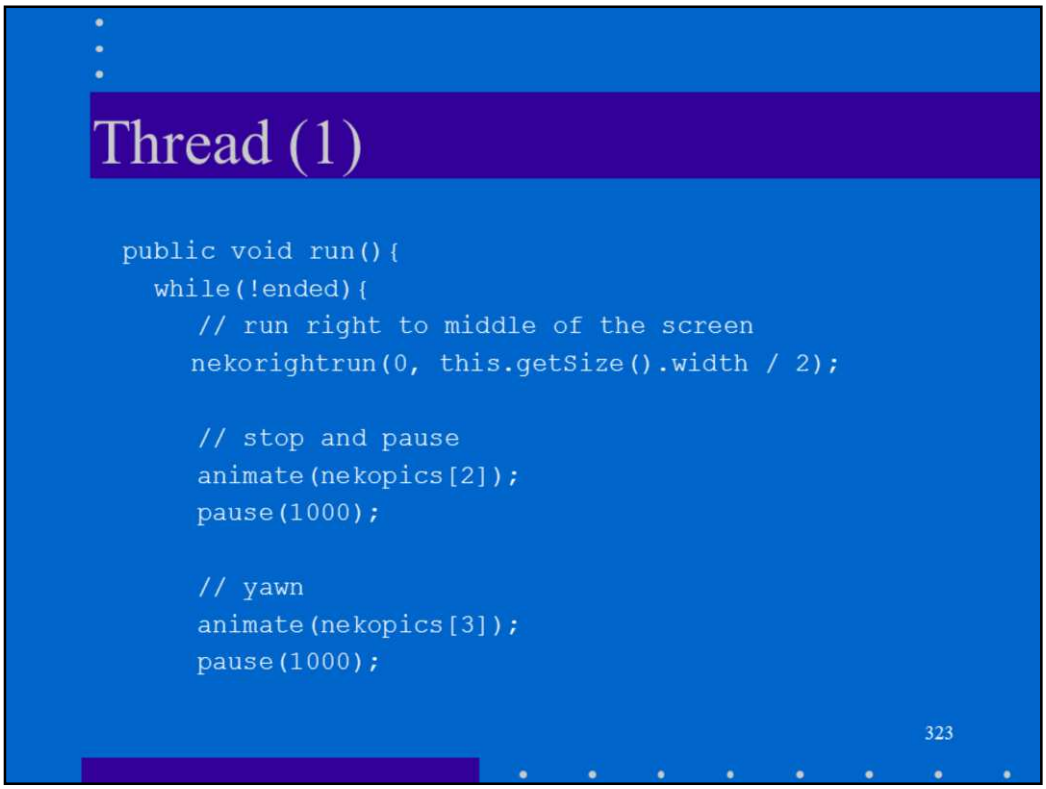
Scratching is shown by alternating the display of the two scratching pictures. This method accepts an argument which controls how many times the cat scratches. Since no change in the x co-ordinate is necessary, the second version of `animate()` can be used.

Sleep

```
void nekosleep(int numtimes) {
   for (int i = numtimes; i > 0; i--) {
     animate(nekopics[6]);
     pause(250);
     animate(nekopics[7]);
     pause(250);
   }
 }
```

322

Sleep works just like scratching except that the sleeping pictures are used.



If a class implements the `Runnable` interface it must have a `run()` method. When an instance of the class is used to create a thread it is the code in the `run()` method which describes what the thread will do.

In this example, the code will repeat until the `ended` flag is set to `true`. This will only ever be set by the applet's `stop()` method.

The cat runs onto the window from the left and runs right to the middle of the window. The left of the screen is pixel position 0. The middle of the window can be found by using the `getSize()` method which class `Applet` inherits from class `Component`. This returns an object of type `Dimension` which has two public variables; `width` and `height`. Half way across the window is half the width value.

The cat stops at this point and so the stop picture is used as the argument to `animate()`. The stop picture is replaced by the yawn picture.

Thread (2)

```
// scratch four times
nekoscratch(4);
// sleep for 5 seconds
nekosleep(5);
// wake up and run off
animate(nekopics[8]);
pause(500);
nekorightrun(xpos, this.getSize().width+10);
//pause a bit
pause(1000);
// run back half way
nekoleftrun(xpos, this.getSize().width/2);
.
.
```

324

The animation proceeds by calling the appropriate action methods or else calling `animate()` directly.

The cat sleeps and then runs off the window to the right. The end point is determined by the size of the window.

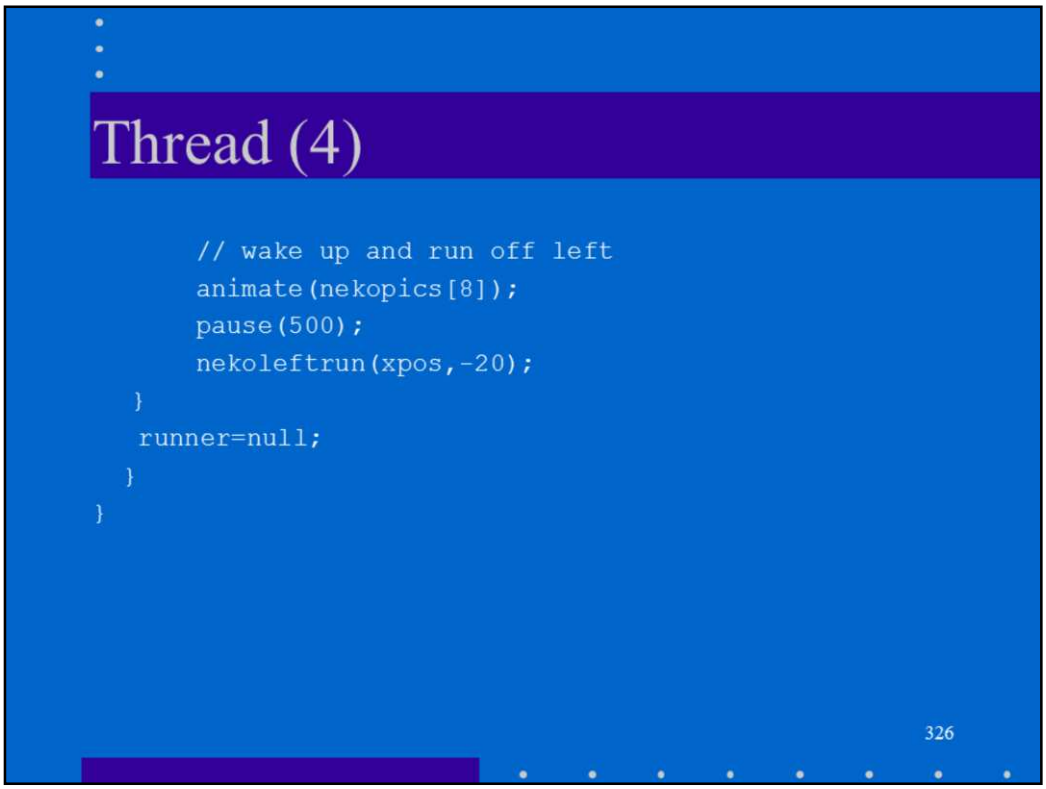
The cat waits a bit and then runs back onto the window to the midpoint.

Thread (3)

```
.
.
// stop and pause
animate(nekopics[2]);
pause(1000);
// yawn
animate(nekopics[3]);
pause(1000);
// scratch ten times
nekoscratch(10);
// sleep two turns
nekosleep(2);
.
.
```

325

Once in the middle it yawns, scratches and sleeps.



Finally the cat runs off the left hand of the window and the sequence begins again.

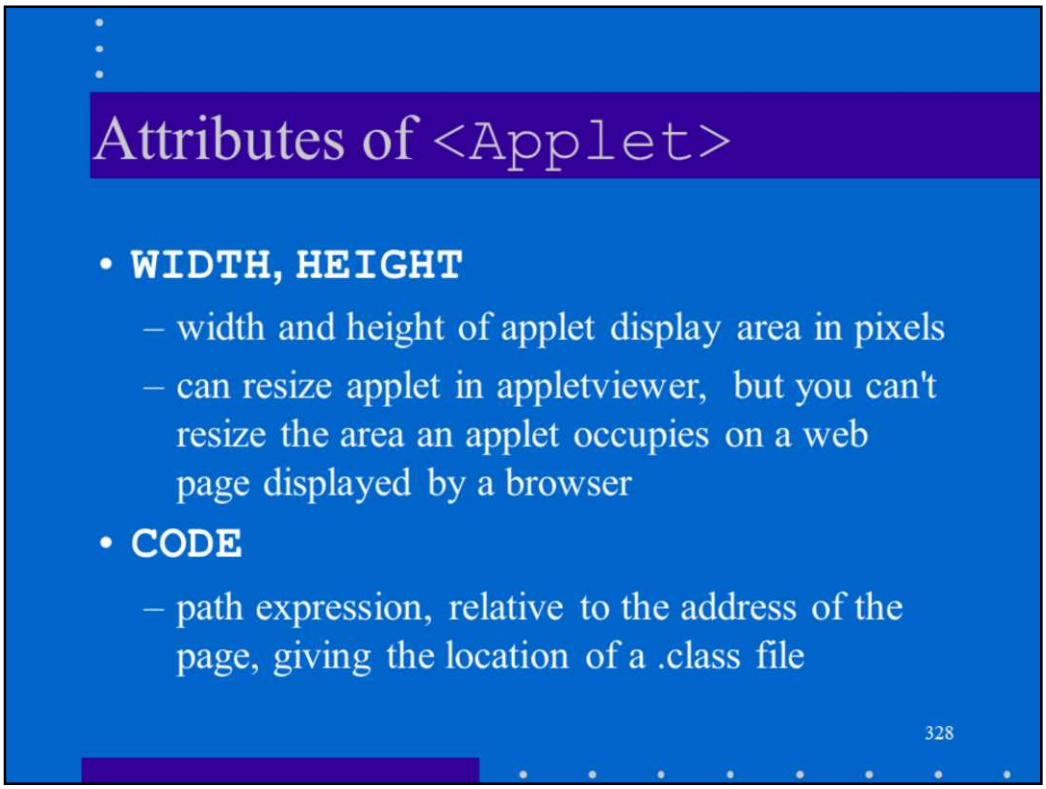
When this applet runs the main action appears in the animation canvas where the thread runs and displays the pictures. Whilst this is going on, the runtime environment is regularly checking to see if either of the buttons have been pressed. As soon as this occurs an event is generated and control is eventually passed to the applet's exception handling routine. At this point the `stopped` flag can be set. This will eventually result in the thread waiting. This mechanism makes it possible for the animation to wait at any point in the sequence and is much less cumbersome than embedding the code to test for a button press in the animation routine.

When the applet's `stop()` method is invoked (usually by the browser when a new Web page is loaded or the image is iconised) the `ended` flag will be set. This will eventually cause the `run()` method of the thread to exit the `while` loop and terminate. Just prior to terminating it sets the pointer the applet uses to reference the thread to `null`. This mechanism ensures that the thread will eventually terminate when the page is not currently in view but does not terminate it immediately. It is left as an exercise for the reader to devise a test to demonstrate that this behaviour does take place (hint: does `start()` always start a new thread?).

HTML for Applets
• HTML - HyperText Markup Language
– the language used to construct WWW pages
– implemented as a set of tags inside angled brackets (<>)
• Basic applet tag <APPLET>
– must specify CODE, WIDTH and HEIGHT
– Has matching </APPLET> tag
327

Applets are meant to be distributed on the World Wide Web. Pages on the Web are coded in a language known as HTML or HyperText Markup Language. HTML is essentially a set of tags which can be inserted into text to instruct a browser how to display it. Tags are surrounded by angled brackets (<>). A few tags such as `<br>` (new line) are stand alone but many tags indicate the start of some attribute which applies to a block of text and therefore have a corresponding tag which marks the end of the block. For example, the tag which marks bold text `<B>` has a corresponding tag `</B>` which indicates the end of a bold section. The string "These are `<B>`Dark `</B>`Deeds" tells the browser to show the word "Dark" in bold but to render the word "Deeds" normally (i.e not bold).

Applets are embedded into HTML using an `<APPLET>` tag. This tag has an associated `</APPLET>` tag. All text between `<APPLET>` and `</APPLET>` is associated with the description of an applet. In addition, the `<APPLET>` tag has some attributes associated with it. Of these, `CODE`, `WIDTH` and `HEIGHT` must be specified.


Attributes of <Applet>
• WIDTH, HEIGHT
– width and height of applet display area in pixels
– can resize applet in appletviewer, but you can't resize the area an applet occupies on a web page displayed by a browser
• CODE
– path expression, relative to the address of the page, giving the location of a .class file
328

The `<APPLET>` tag must be supplied with values for `CODE`, `WIDTH` and `HEIGHT`:

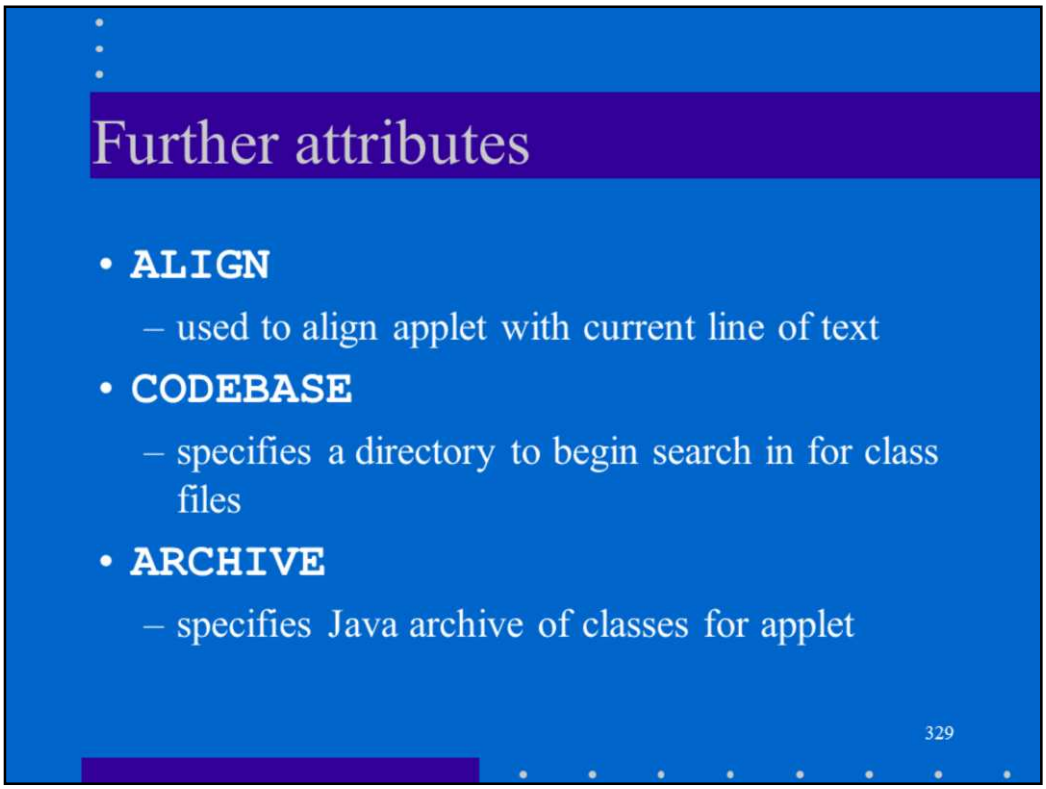
```
<APPLET CODE="MyApplet.class", HEIGHT=150,
WIDTH=50></APPLET>
```

The width and height attributes tell the browser how much space to allocate for the visual representation of the applet. In other words they specify the window size for the applet. An applet running in the applet viewer can have its window resized, but this is not possible when the applet runs within a browser.

The code attribute tells the browser where to load the Java byte codes from. This will be the name of a class file. Obviously, this reference to a class file occurs in some HTML which is stored in a file. If the `CODE` attribute is set to a file name then the browser tries to load the byte codes from a file in the same directory that the HTML page came from. The `CODE` attribute may, however, contain directory information in which case the browser looks for directories relative to the one from which the HTML page came.

Suppose an HTML page is stored in directory /home/public_html. If this page has an applet tag with `CODE` set to `myapps/MyApplet.class` then the browser will try to load `/home/public_html/myapps/MyApplet.class`.

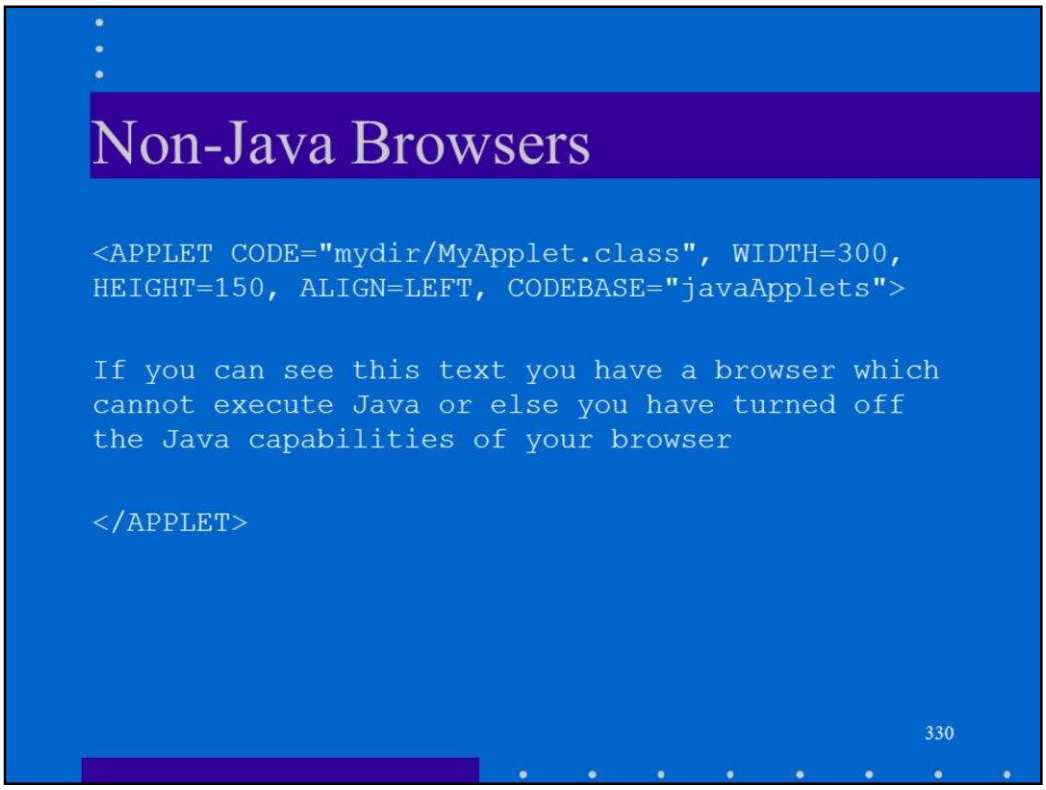
See the `CODEBASE` attribute (described later) to see how this behaviour can be modified.


Further attributes
• ALIGN
– used to align applet with current line of text
• CODEBASE
– specifies a directory to begin search in for class files
• ARCHIVE
– specifies Java archive of classes for applet
329

The `ALIGN` attribute lines up the position of the attribute with the current line of text. Valid options are `LEFT`, `RIGHT`, `BOTTOM`, `TOP`, `TEXTTOP`, `MIDDLE`, `ABSMIDDLE`, `BASELINE` and `ABSBOTTOM`.

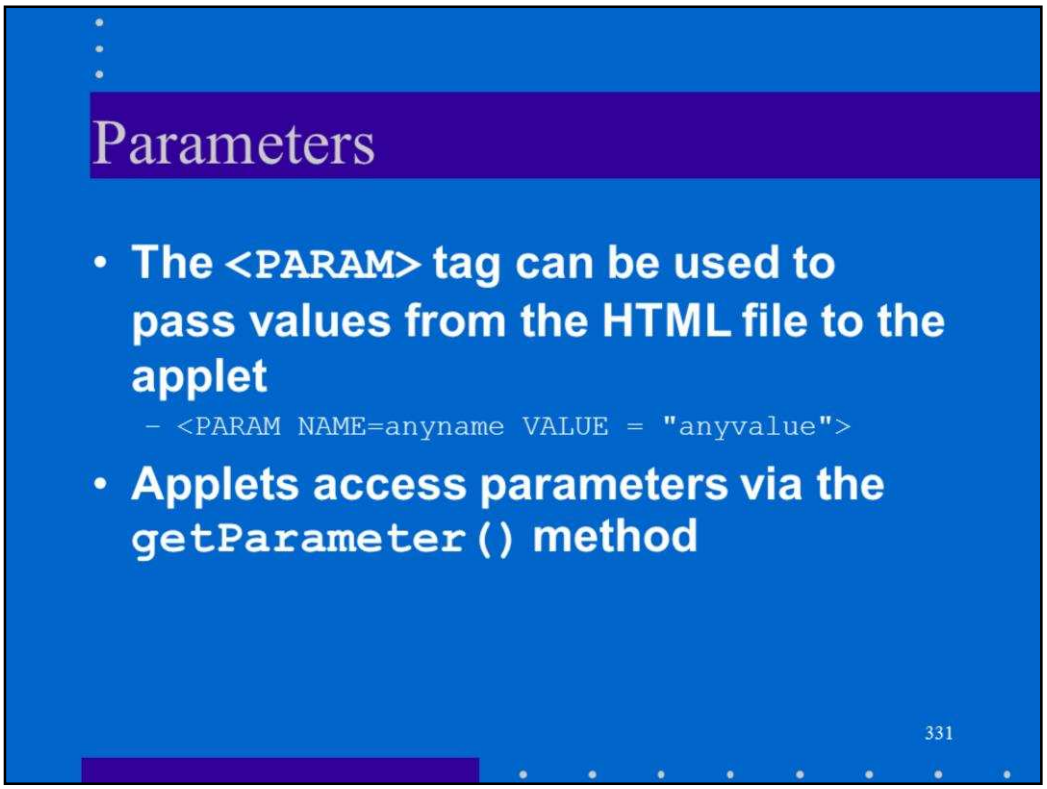
`CODEBASE` is used to specify a directory path in which the browser should search to find the Java byte codes. The top level directory of this path must be located in the directory where the HTML was retrieved from.

Java programs quite frequently rely on the existence of many other classes. The standard classes are available to Java enabled browsers. It may be, however, that an applet requires a number of user written classes. Normally these will be downloaded on demand. To speed up the application execution process it is possible to preload classes. The `ARCHIVE` tag can be used to specify Java archive files (JAR files) which must be loaded before the applet executes.



This tag will cause the browser to allocate a window 300 pixels wide and 150 pixels high in the left hand margin of the page it is displaying. It will then fetch and execute a file called MyApplet.class. The class will be located by looking in the javapplets subdirectory of the directory from which the HTML was retrieved. This subdirectory is searched for the directory mydir and the class file should be located there.

If the browser cannot cope with Java or the user has turned off the browser's Java handling mechanism then the text which appears between the `<APPLET>` and the `</APPLET>` tags is displayed instead. It's a good idea to include text of this type as it instantly informs a reader that they are missing something.



The other item that may appear between the `<APPLET>` tag and the `</APPLET>` tag is any number of `<PARAM>` tags. These can be used to pass parameter information from the surrounding HTML to the applet. It is therefore possible to vary the behaviour of the applet in a manner which depends on the HTML it is embedded in. This is particularly useful in enabling a programmer to write a generalised applet for displaying scrolling text. The text to be scrolled can be specified by the HTML writer rather than the Java programmer.

The `<PARAM>` tag has two attributes; `NAME` names the parameter, `VALUE` assigns the parameter a value of type `String`.

Applets use the `getParameter()` method to access the values of parameters.

```
public String getParameter(String name)
```

Returns the value of the named parameter in the HTML tag. For example, if this applet is specified as

```
<applet code="Clock" width=50 height=50>
<param name=Colour value="blue">
</applet>
```

then a call to `getParameter("Colour")` returns the value "blue".

The name argument is case insensitive.

Parameter Example

```
import java.applet.*;
import java.awt.Graphics;

public class ParameterExample extends Applet{

        public void paint(Graphics g){
                String param;

                param=getParameter("text");
                g.drawString(param,25,25);
        }
}
```

332

In this example, an applet retrieves a string from the parameter called text and draws that string at pixel position 25, 25.

HTML

```
<html>
<head>
<title>Parameter Example</title>
</head>
<body>
<applet code="ParameterExample.class"
        height=100 width=100>
<param name=text value="Bye Bye">
<p>Your browser does not support Java</p>
</applet>
</body>
</html>
```

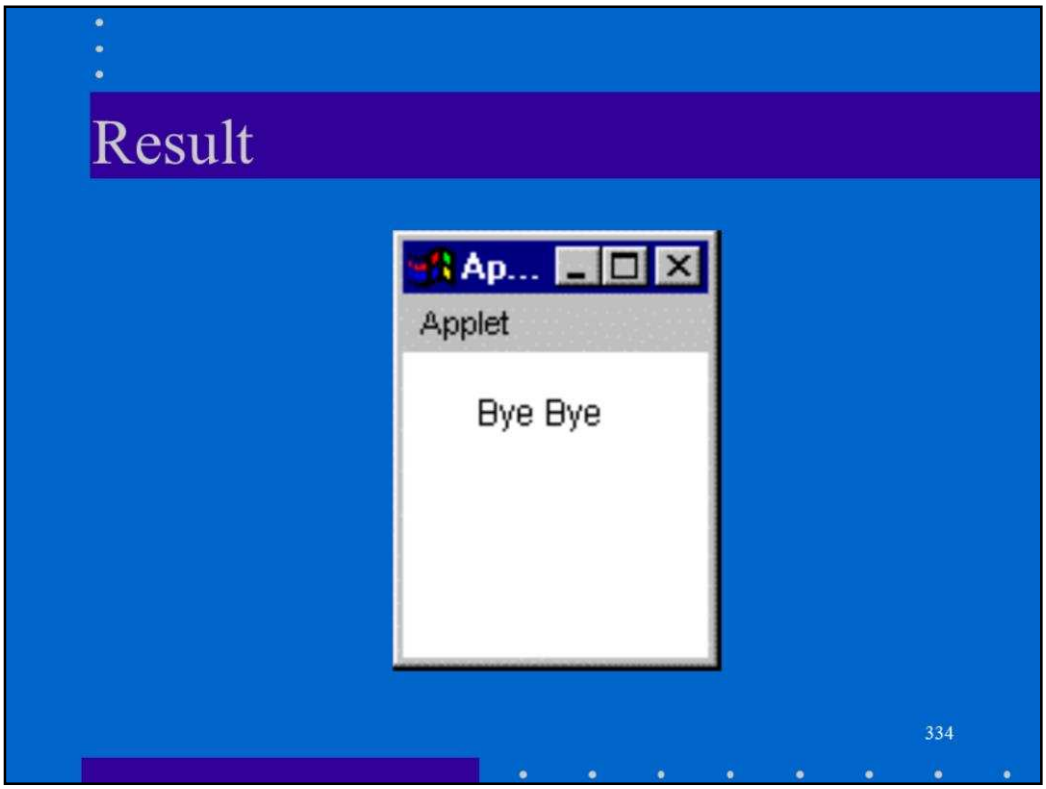
333

This is the HTML used to execute the applet. The applet is located in the file ParameterExample.class and is allocated a window space 100 pixels high and 100 pixels wide. A parameter named text is assigned a value "Bye Bye".

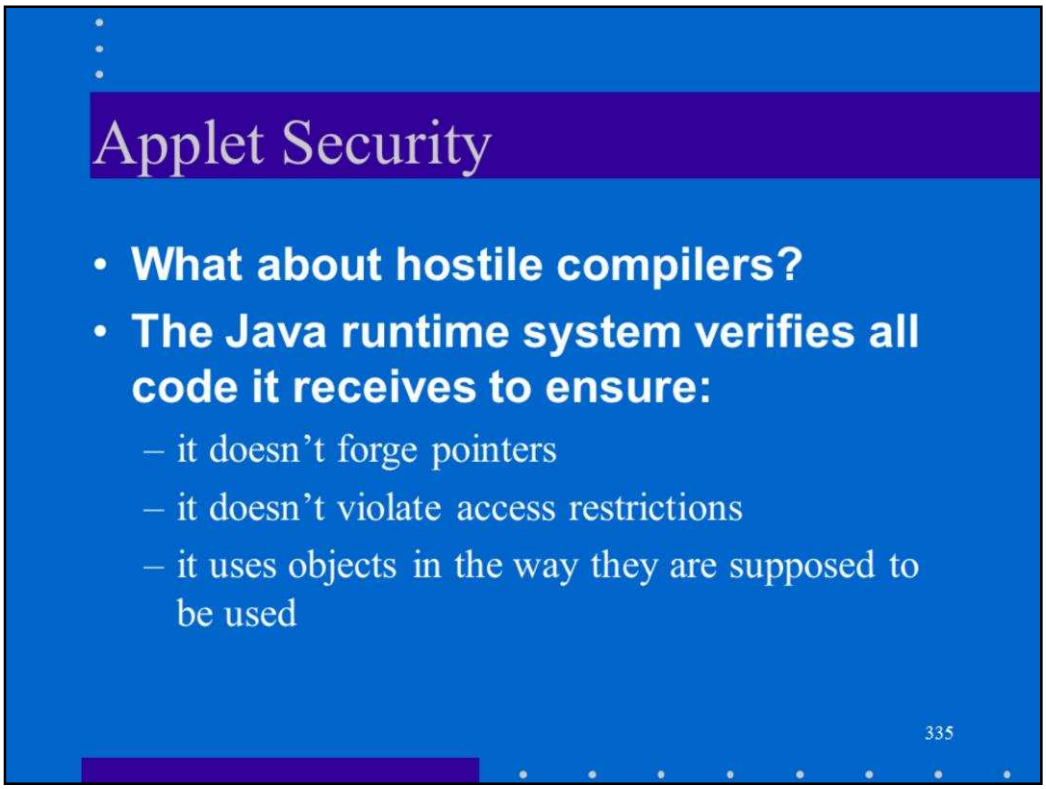
The applet will read this parameter and display the text in the window.

Full details of all valid HTML tags are available at:

http://www.scit.wlv.ac.uk/encyc/



This is the result of invoking the previous HTML file with the appletviewer. From this demonstration it should be clear that is relatively easy to write an applet to display text so that the text and its position can be specified by the HTML file that references it. This opens the way to a number of generalised applets which can deliver results that are tailored by the HTML in which they are embedded.

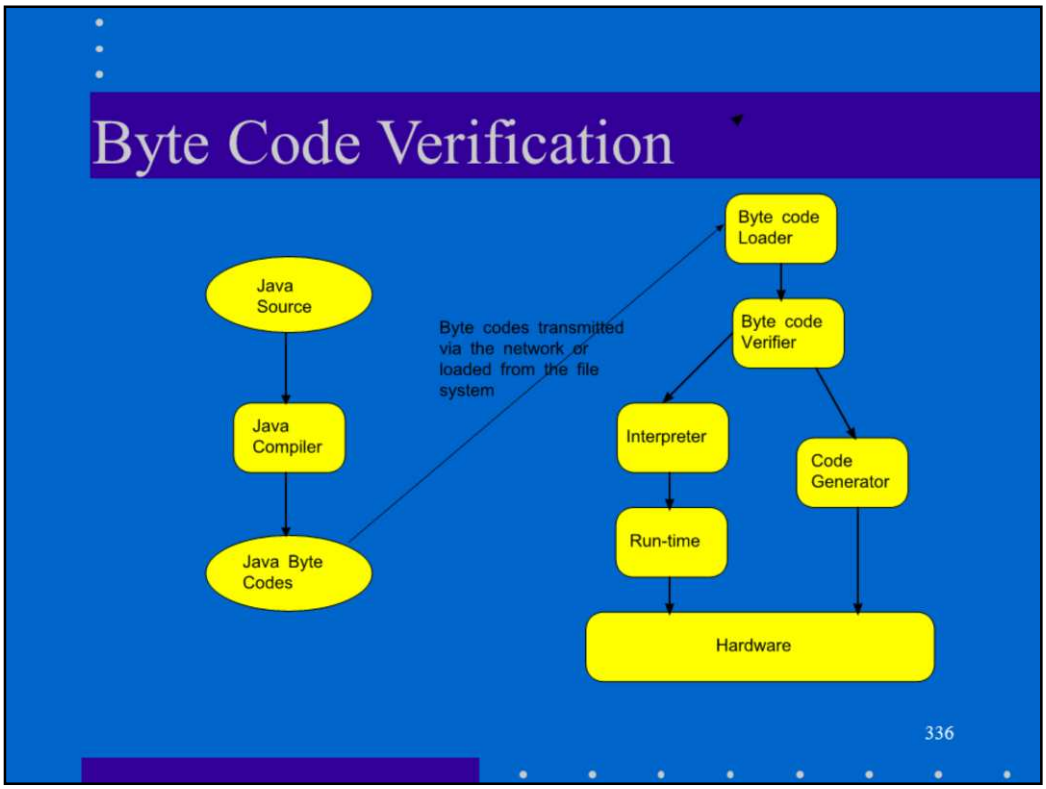

Applet Security
• What about hostile compilers?
• The Java runtime system verifies all code it receives to ensure:
– it doesn't forge pointers
– it doesn't violate access restrictions
– it uses objects in the way they are supposed to be used
335

One of the inhibitions to the widespread use of the Internet has been the fear that it is full of software which can spread viruses to any computer connected to it. Applets make things potentially more worrying. They are pieces of software which come from the Internet and run on your machine. Due to Java's cross platform features they will happily run on virtually any hardware and operating system combination.

As it stands, the compiler that is supplied with the JDK is relatively safe. It can't be used to generate harmful programs. However, it is possible to conceive of a compiler which would allow a malicious person to compile an applet capable of sabotaging the machine of anyone that runs it.

To prevent this and allay fears of this kind, the Java run time environment has been constructed so as to prevent hostile applets running on a machine. It has a number of mechanisms for disabling applets and applications which behave in a hostile manner.

A discussion of Java security problems and their solutions can be found at: http://www.ibm.com/java/education/whatprice.html



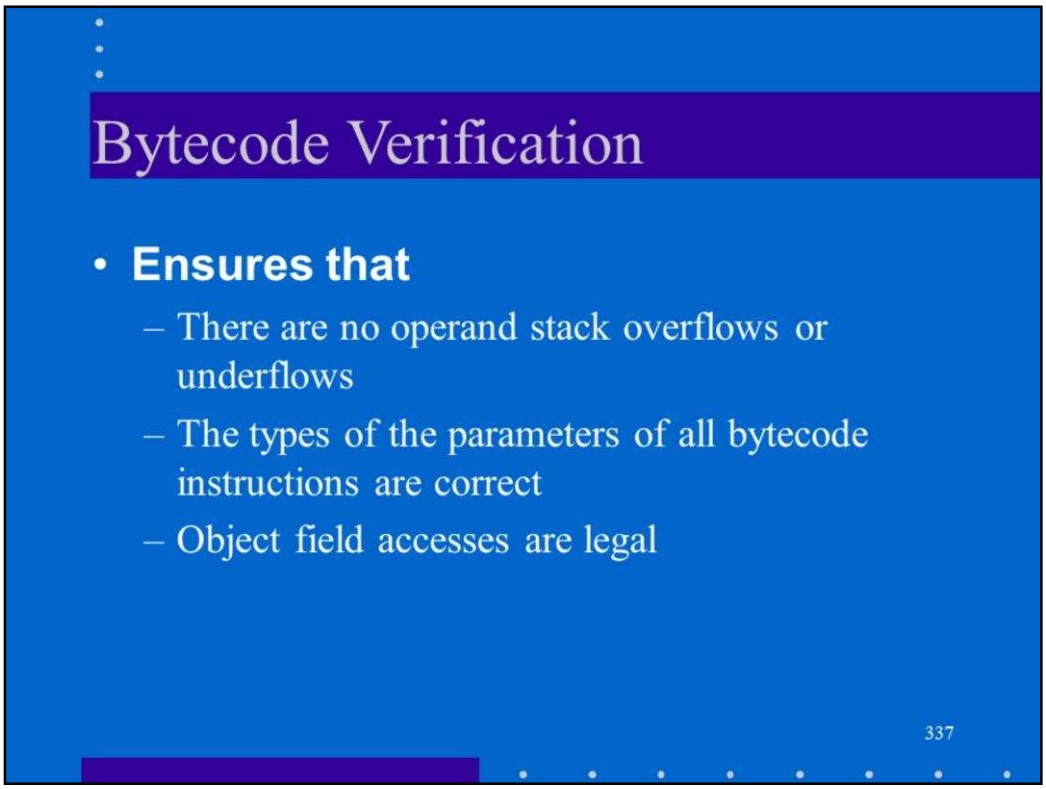
The Java byte codes which are the output of the Java compilation process are checked carefully by a machine which receives them in the form of an applet.

The Java byte code loader, determines how (and when) applets can load classes (code). One of its more important functions is to ensure that running applets do not replace system-level components within the runtime environment. This component is typically supplied by the browser vendor--although it is usually based on templates supplied by Sun Microsystems.

Secondly, all byte codes are verified, this:

• Ensures that the compiled code is formatted correctly.

• Applies a theorem-prover to each member function to ensure that its byte codes do not violate access type restrictions or pointer addressing rules


Bytecode Verification
• Ensures that
– There are no operand stack overflows or underflows
– The types of the parameters of all bytecode instructions are correct
– Object field accesses are legal
337

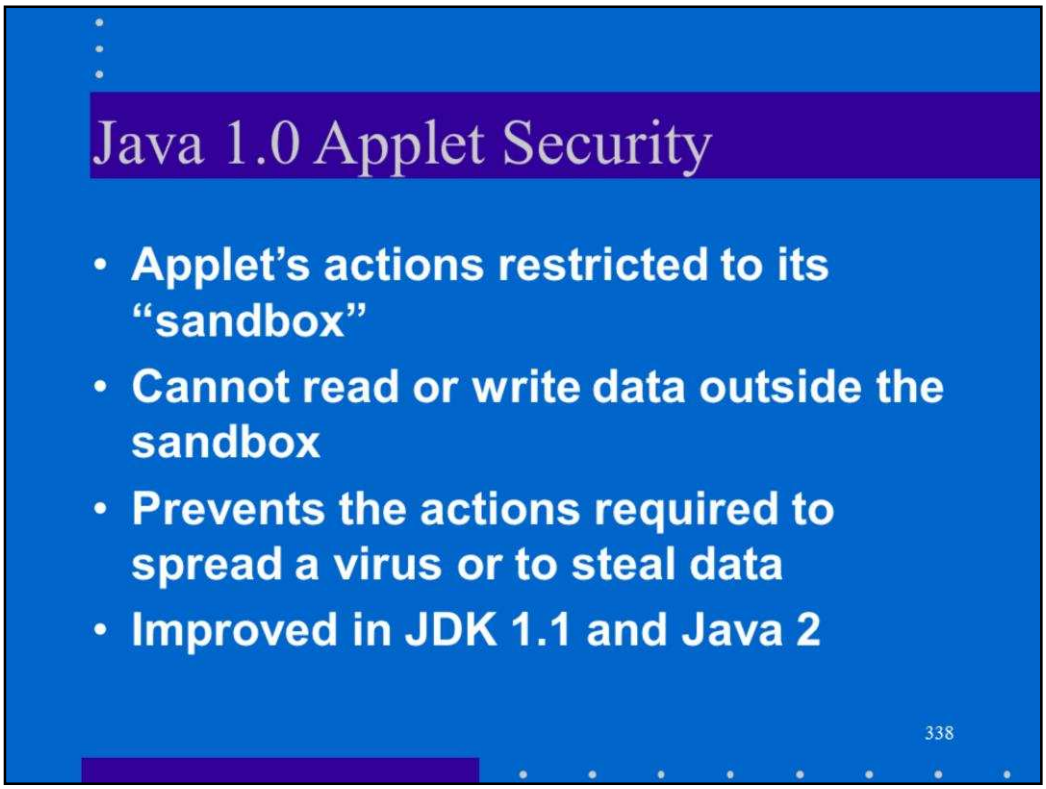
After verification the following is guaranteed:

• Internal stacks will not overflow or underflow. (When either of these events occurs, systems are unstable and hackers can "play in the confusion.") This type of attack has led the way to some of the most infamous security breaches.

• No illegal data conversions will occur (i.e., the verifier will not allow integers to serve as pointers). This ensures that variables will not be granted access to restricted memory areas.

• Byte-code instructions will have appropriately typed parameters (for the same reason as stated above).

• Within an object, only public data and functions can be accessed. Access to all components is restricted. As a result, the proprietary data remains private.


Java 1.0 Applet Security
• Applet's actions restricted to its "sandbox"
• Cannot read or write data outside the sandbox
• Prevents the actions required to spread a virus or to steal data
• Improved in JDK 1.1 and Java 2
338

The third and most important component of Java's security model is the Security Manager. Its task is to perform runtime verification of all so-called "dangerous methods," that is, methods that request file I/O, network access, or want to define a new Class Loader. Some of its duties include:

Managing all network operations.

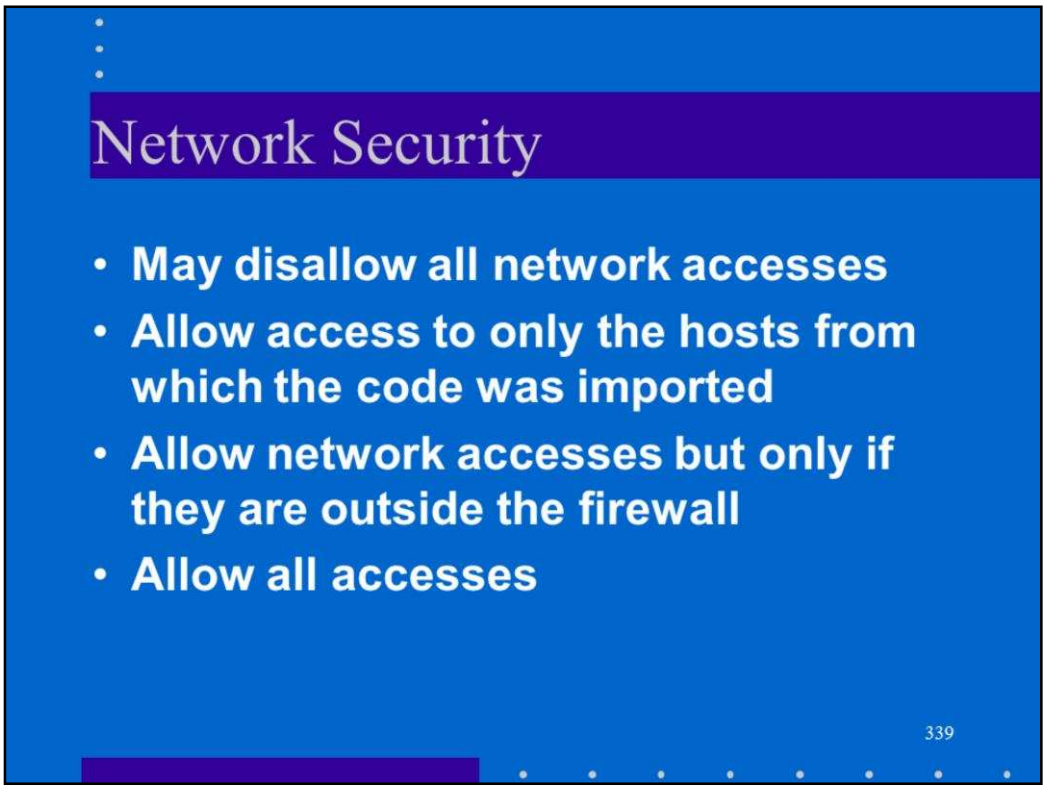
Controlling the creation of, and all access to, operating system processes.

Preventing the installation of new Class Loaders.

Managing file access (obviously, we do not want applets reading files at will).

To ensure compliance, all methods that are part of the basic Java libraries (i.e., those supplied by Sun) consult the Security Manager prior to executing any dangerous operations. In all such cases, the Security Manager may exercise veto power over any request.

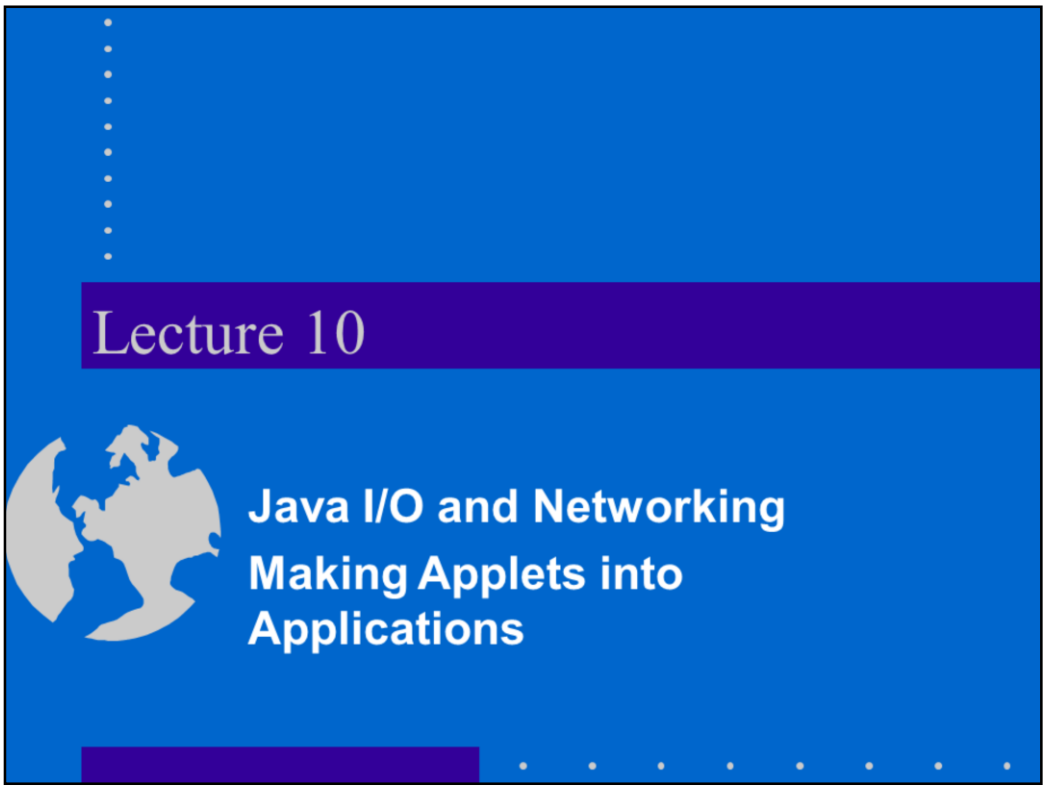
The JDK 1.1 allowed 'signed' applets access outside of the sandbox. Java 2 refines this to allow users to specify the allowed modes of access a signed applet may have.


Network Security
• May disallow all network accesses
• Allow access to only the hosts from which the code was imported
• Allow network accesses but only if they are outside the firewall
• Allow all accesses
339

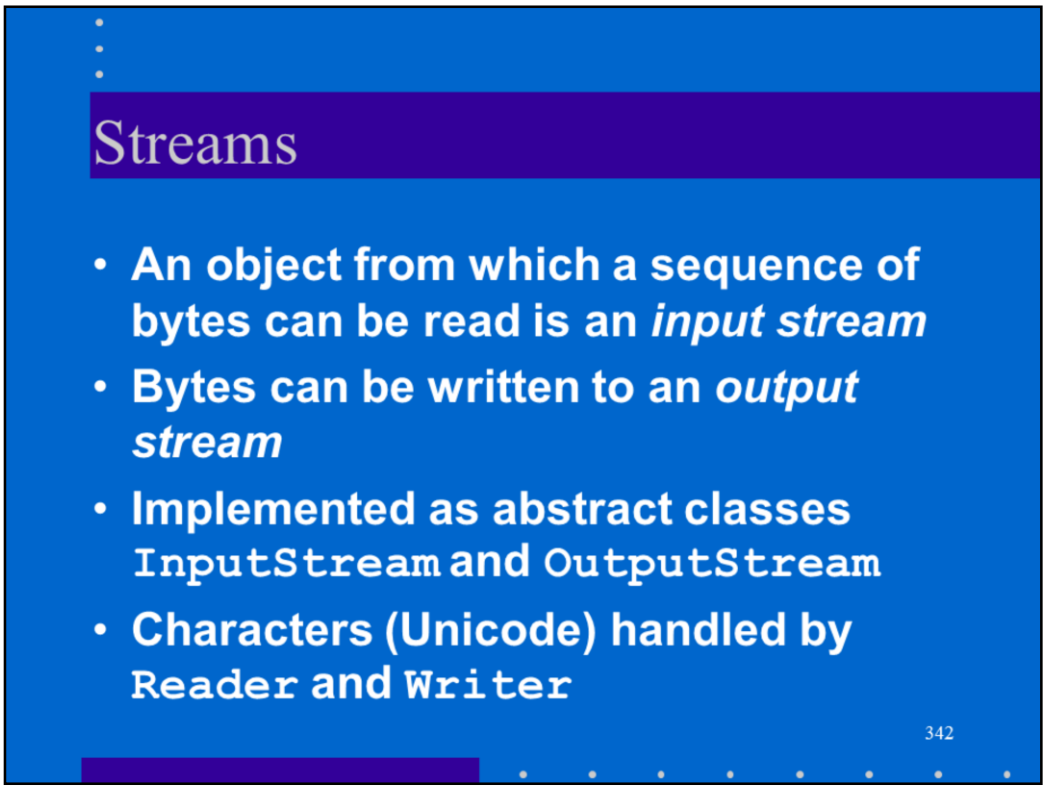
One further problem that might occur with a Java applet is that whilst it might not itself constitute a virus it might be able to download one from the Internet. Java allows system administrators to control the sort of access a Java applet has to the the Internet. The normal restriction is that code can only be down loaded from the machine where the original applet came from. This can cause difficulties for people who wish to construct distributed Java applications but it is considered to be necessary to assuage security fears.

Summary

- **Runnable Interface**
- **Animation example**
  - Threads
  - Events
- **HTML for Applets**
  - <APPLET> </APPLET>
  - <PARAM>
- **Applet Security**



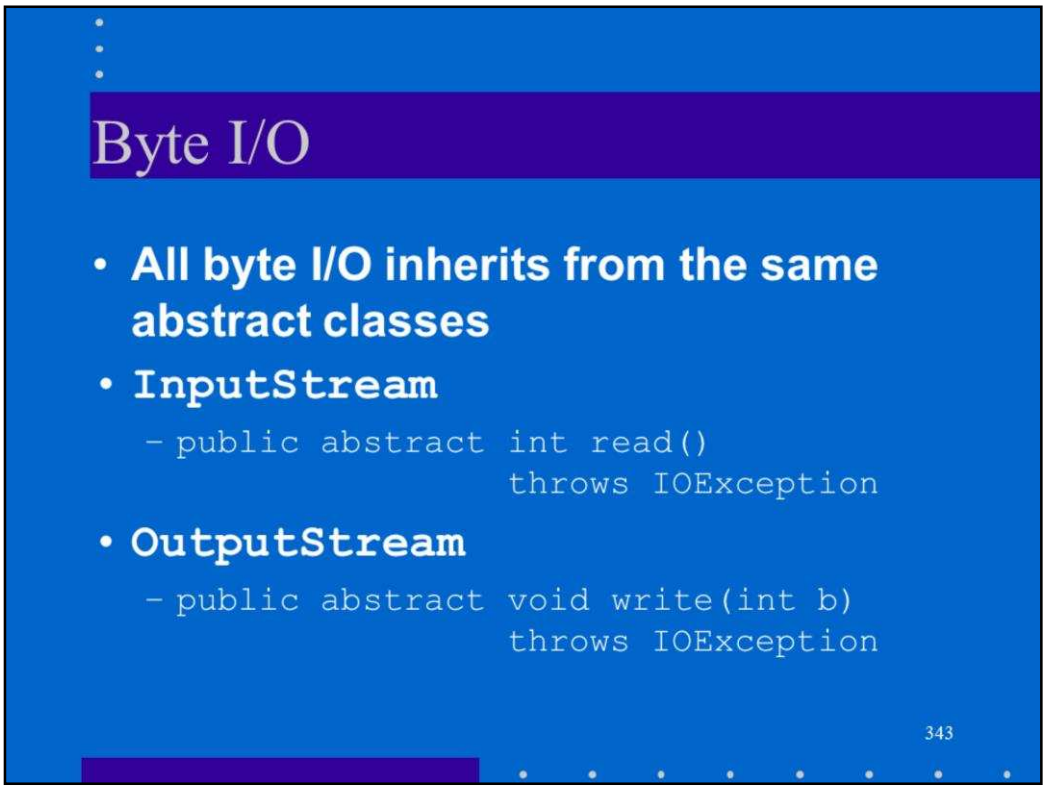
The main topic of this lecture is the way Java deals with network connections. Java is billed as being "the language of the Internet" and therefore one would expect to find powerful facilities for connecting between computers. In fact Java is even more subtle than this because it treats a network link as though it were a conventional input stream. A prerequisite to understanding how networking is achieved is an understanding of how Java deals with conventional input and output.


Streams
• An object from which a sequence of bytes can be read is an input stream
• Bytes can be written to an output stream
• Implemented as abstract classes InputStream and OutputStream
• Characters (Unicode) handled by Reader and Writer
342

Java I/O is based on the concept of streams. This is the technique used in C++. A stream is a connection between a source or a sink and a program. Bytes flow from the source one at a time into the program and from the program to the sink in the same way. A stream providing input is an *input stream*. A stream accepting output is an *output stream*.

There are two abstract classes associated with I/O streams. As with all abstract classes they collect in one place all the essential functions that a stream should implement. The two classes are `InputStream` and `OutputStream`. as we shall see these two classes stand at the top of two large hierarchies of streams. The philosophy of Java is to provide a vast collection of streams so that the compiler can check that a user is not accidentally misusing a stream by carrying out an illegal operation on it. C in contrast has one stream type `FILE*`. Visual Basic has three. C++ has more stream classes than these languages but not as many as Java 1.1 which has 58!

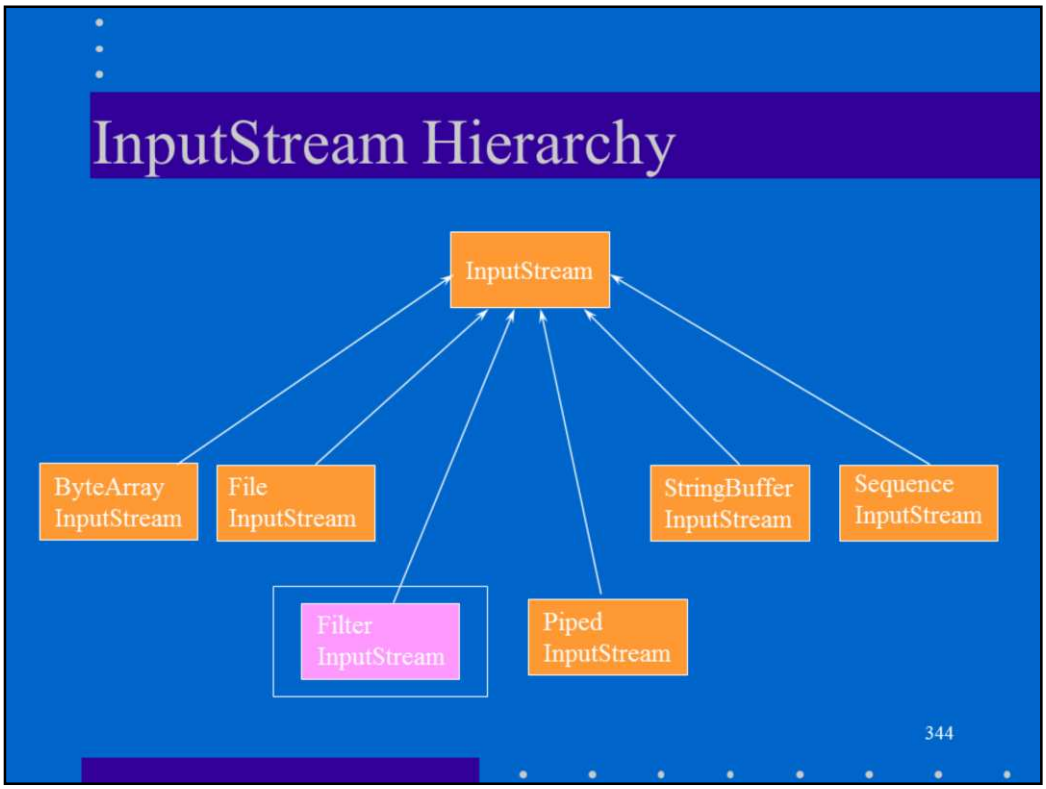
Classes which inherit from `InputStream` or `OutputStream` are based on single byte input or output. If Java used standard ASCII this would be sufficient for character I/O. However in Java, characters are represented by 16 bit Unicode characters. For this reason two further abstract classes `Reader` and `Writer` and their associated hierarchies are required to implement character I/O.



The class hierarchies for I/O make heavy use of overriding. Classes `InputStream` and `OutputStream` contain many methods which are not abstract and can be used by many of the other I/O classes. All of these methods, however, rely on the implementation of `read()` in the case of an input stream, or `write()` for output streams. Subclasses provide their own peculiar implementations of `read()` and `write()` to create new types of streams.

The definition of `read()` is that it reads one byte from a stream, returns the byte read or -1 if the end of the input source has been reached.

`write()` writes a single byte to an output stream.



This slide shows the hierarchy headed by `InputStream`. All of these appear in the package java.io.

```
public abstract class InputStream
```

This abstract class is the superclass of all classes representing an input stream of bytes.

Each subclass provides facilities for a special type of input stream, for example:

```
public class ByteArrayInputStream
```

This class allows an application to create an input stream in which the bytes read are supplied by the contents of a byte array. Applications can also read bytes from a string by using a `StringBufferInputStream`.

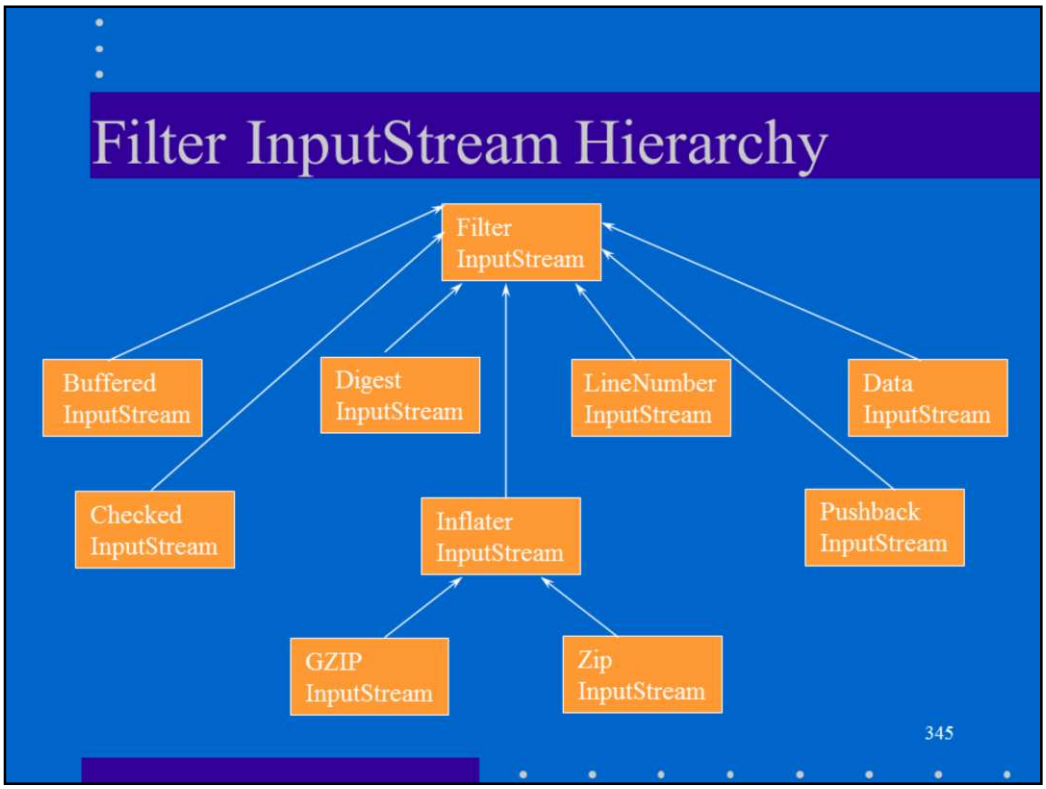
```
public class FileInputStream
```

A file input stream is an input stream for reading data from a `File`.

```
public class CheckedInputStream
```

An input stream that also maintains a checksum of the data being read. The checksum can then be used to verify the integrity of the input data.

Full details of these classes can be found in the online Java documentation.



`public class FilterInputStream extends InputStream`

This class is the superclass of all classes that filter input streams. These streams sit on top of an already existing input stream (the underlying input stream), but provide additional functionality.

The class `FilterInputStream` itself simply overrides all methods of `InputStream` with versions that pass all requests to the underlying input stream. Subclasses of `FilterInputStream` may further override some of these methods as well as provide additional methods and fields.

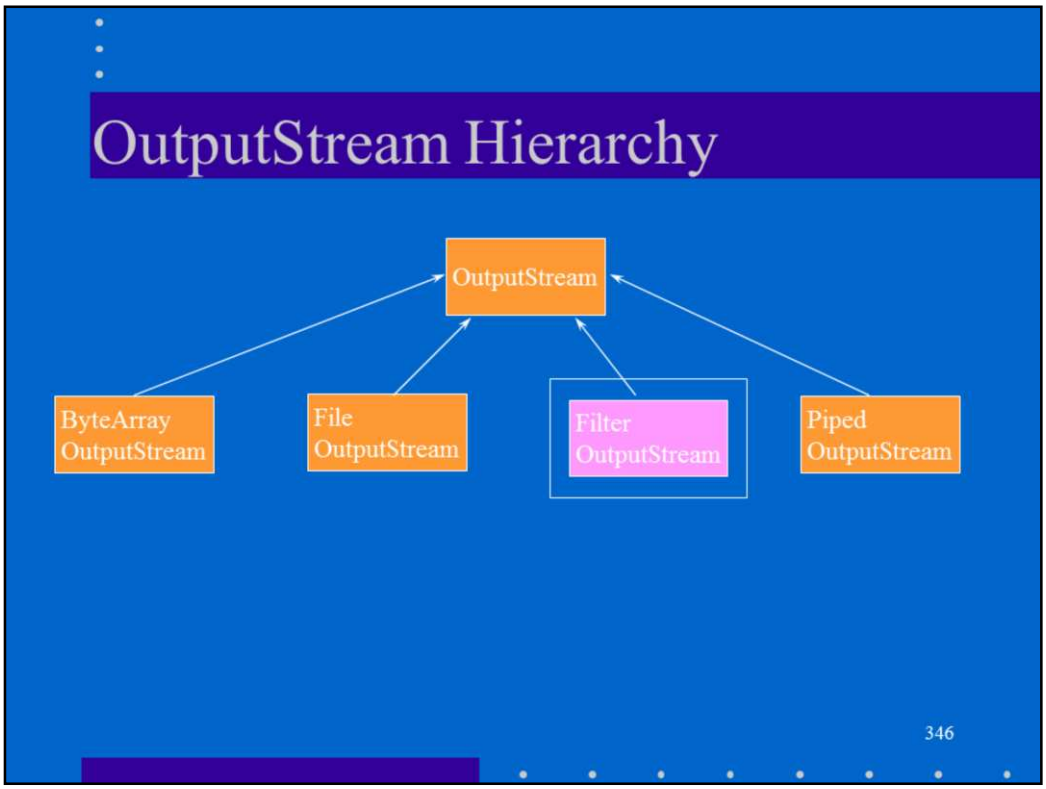
`public class BufferedInputStream`

The class implements a buffered input stream. By setting up such an input stream, an application can read bytes from a stream without necessarily causing a call to the underlying system for each byte read. The data is read by blocks into a buffer; subsequent reads can access the data directly from the buffer.

`public class DataInputStream`

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

Full details in the online Java documentation



This slide shows the hierarchy headed by `OutputStream`. All of these appear in the package java.io.

`public class ByteArrayOutputStream`

This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using `toByteArray()` and `toString()`.

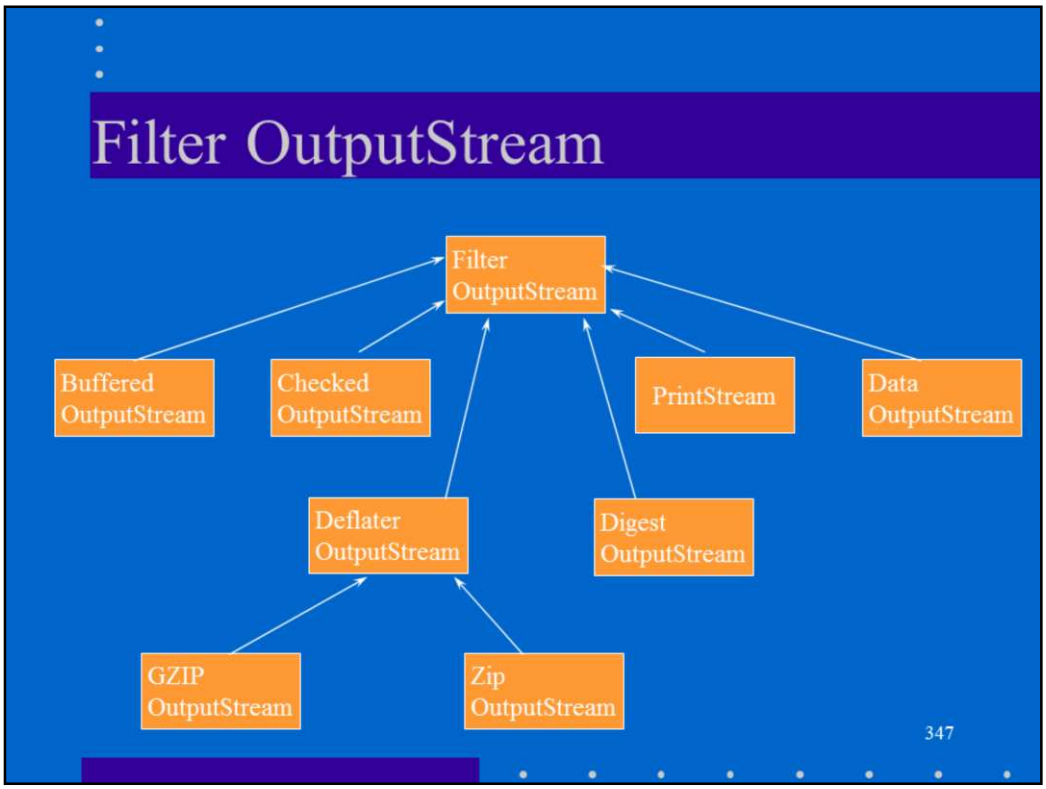
`public class FileOutputStream`

A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`.

`public class PipedOutputStream`

A piped output stream is the sending end of a communications pipe. Two threads can communicate by having one thread send data through a piped output stream and having the other thread read the data through a piped input stream.

Full details in the online Java documentation.



`public class FilterOutputStream extends OutputStream`

This class is the superclass of all classes that filter output streams. These streams sit on top of an already existing output stream (the underlying output stream), but provide additional functionality.

The class `FilterOutputStream` itself simply overrides all methods of `OutputStream` with versions that pass all requests to the underlying output stream. Subclasses of `FilterOutputStream` may further override some of these methods as well as provide additional methods and fields.

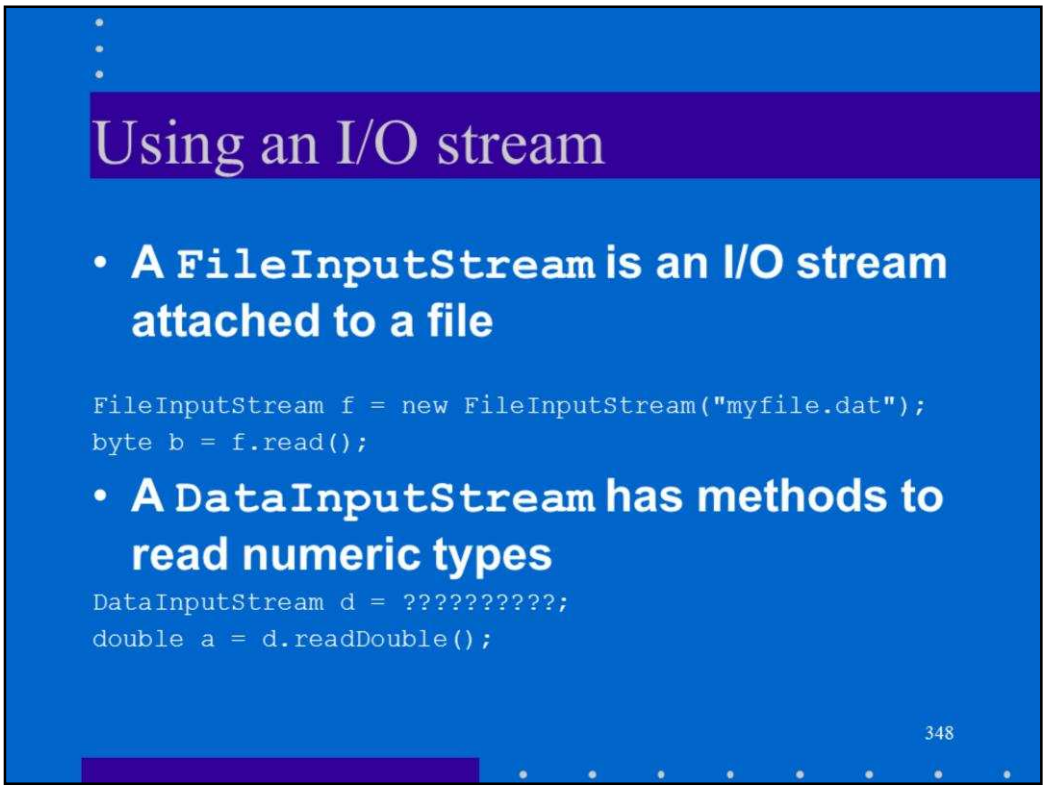
`public class BufferedOutputStream`

The class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written. The data is written into a buffer, and then written to the underlying stream if the buffer reaches its capacity, or the buffer output stream is closed, or the buffer output stream is explicitly flushed.

`public class DataOutputStream`

A data input stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

Most input stream classes have a matching output stream class.

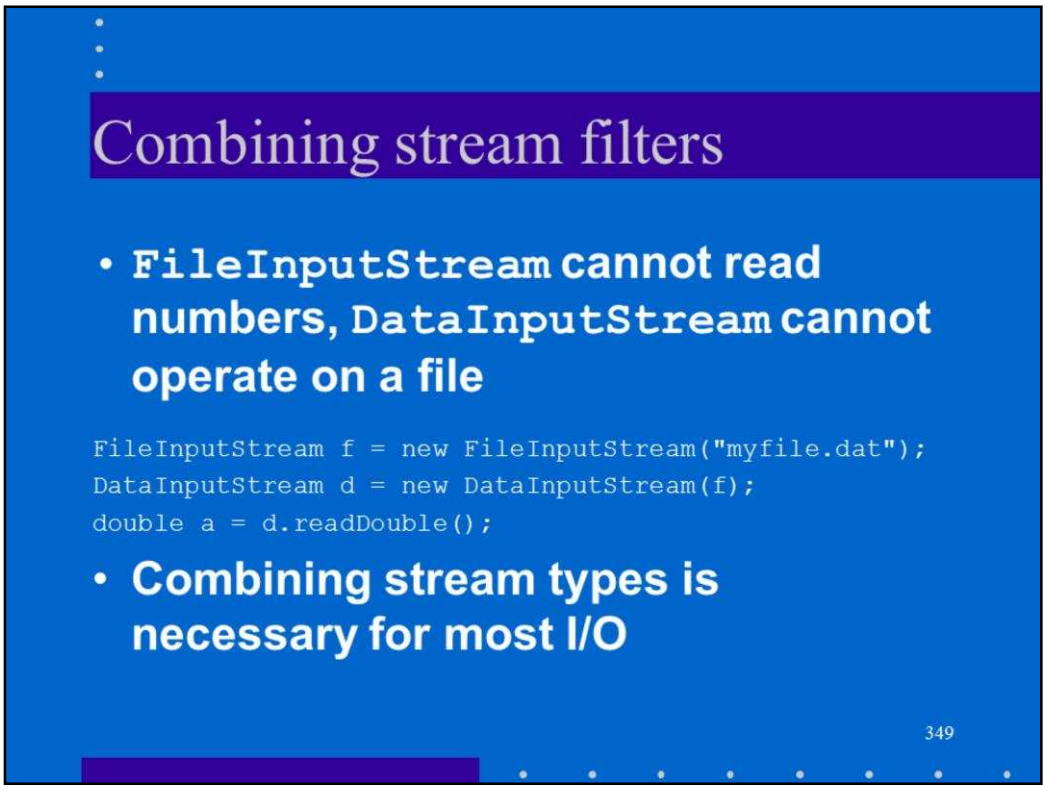

Using an I/O stream
• A FileInputStream is an I/O stream attached to a file
FileInputStream f = new FileInputStream("myfile.dat");
byte b = f.read();
• A DataInputStream has methods to read numeric types
DataInputStream d = ??????????;
double a = d.readDouble();
348

The normal way to use Java's input streams is to use a class which is a direct descendant of `InputStream` to connect to the stream and then use a subclass of `FilterInputStream` to add functionality to the stream.

For example, a `FileInputStream` has a constructor which allows a programmer to create a stream which reads from a named file. The only I/O operation supported by `FileInputStream` is however, `read()` which inputs a single byte.

On the other hand, a `DataInputStream` has methods which will read all the basic Java types from a file but lacks a mechanism to connect to a file.

On their own, neither of these stream classes are much use!


Combining stream filters
• FileInputStream cannot read numbers, DataInputStream cannot operate on a file
FileInputStream f = new FileInputStream("myfile.dat");
DataInputStream d = new DataInputStream(f);
double a = d.readDouble();
• Combining stream types is necessary for most I/O
349

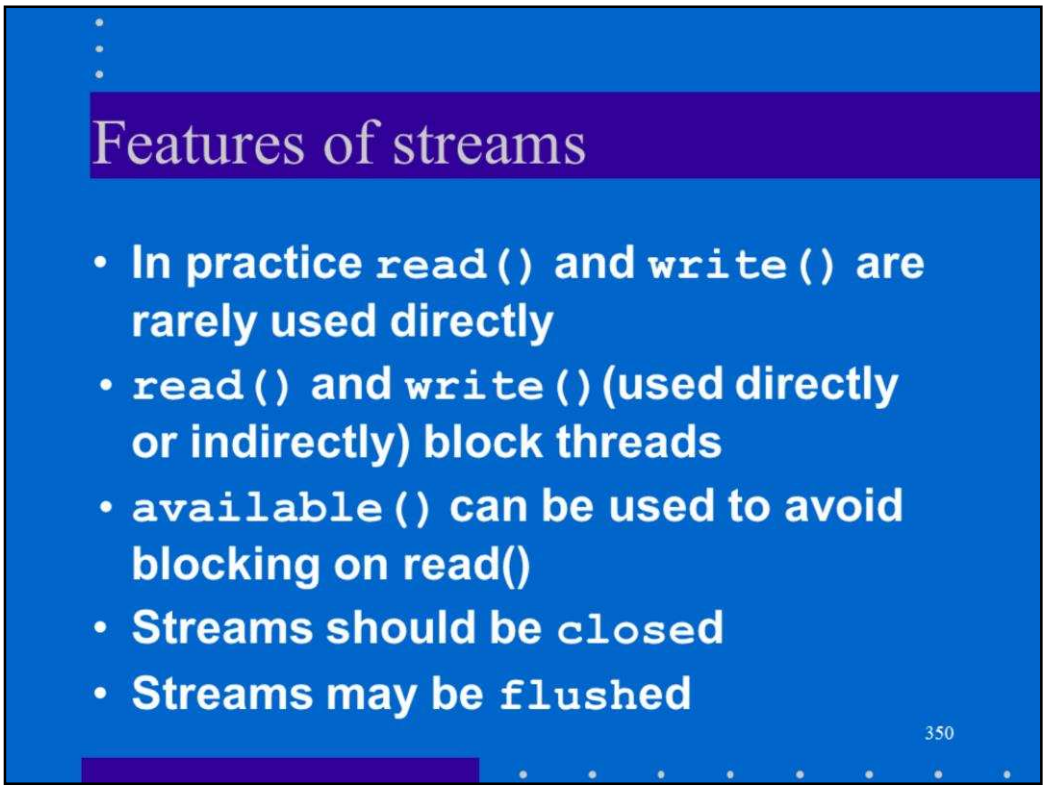
The way to deal with this is to combine the two types of streams. The constructor for `FilterInputStream` and its subclasses expects an `InputStream` as its parameter.

The code fragment on this slide shows how a `FileInputStream` is opened on myfile.dat. This is then used to create a `DataInputStream`. The result is a stream which reads from myfile.dat and responds to the methods of class `DataInputStream`.

The code could be rewritten:

```
DataInputStream d=new
 DataInputStream(new
FileInputStream("myfile.dat"));
```

This saves the need for one temporary variable, but is less readable and prevents the programmer accessing the underlying `FileInputStream`.


Features of streams
• In practice read() and write() are rarely used directly
• read() and write() (used directly or indirectly) block threads
• available() can be used to avoid blocking on read()
• Streams should be closed
• Streams may be flushed
350

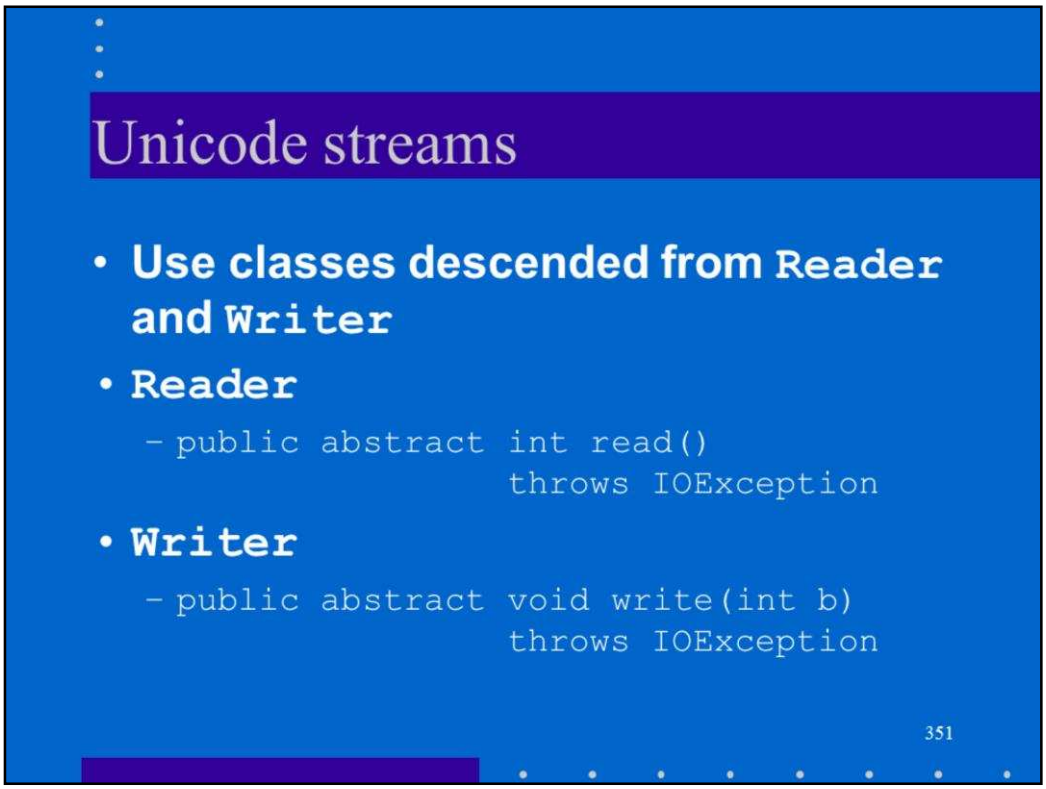
The mechanism shown on the previous slide means that methods `read()` and `write()` are rarely used explicitly in programs. They are usually invoked indirectly by methods of a subclass of `InputStream` or `OutputStream`.

Regardless of whether they are explicitly called or not, calls to `read()` and `write()` cause a thread to block.

In the case of an input operation a thread which calls `read()` may block because the data it requires is not yet available. The method `available()` returns an `int` which gives the number of bytes which are available for reading. Threads that wish to avoid waiting for input can make use of this method by not invoking read() when there are no bytes available..

All streams require buffer space etc. When a stream is no longer required this buffer space can be released. All I/O streams implement a method `close()` which will achieve this.

In order to optimise disk I/O, streams will buffer data. Typically an output stream will hold data in main memory until enough has been generated to make it worthwhile to write to disk. The programmer can cause buffers to be emptied on demand by invoking the method `flush()`.

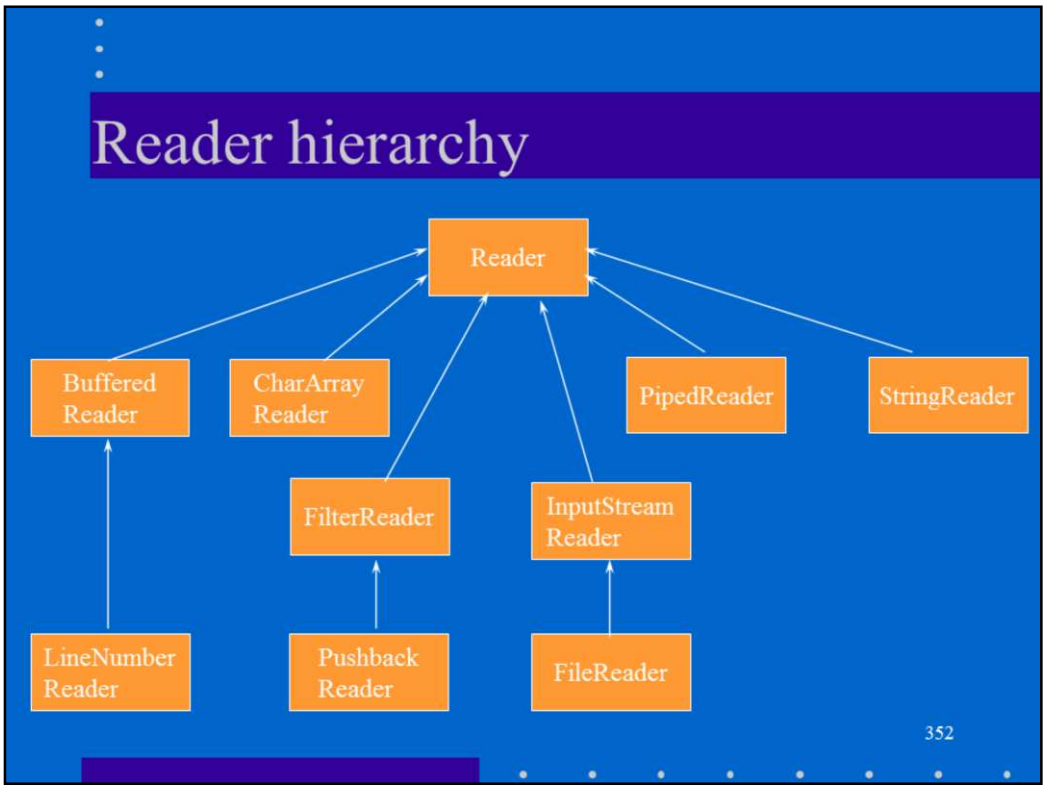


The classes `InputStream` and `OutputStream` deal with byte input and output. The classes `Reader` and `Writer` are provided to read two byte Unicode characters. `Reader` and `Writer` are abstract classes at the top of two large hierarchies.

`read()` returns an integer between 1 and 65535, or -1 if the send of the stream has been reached.

`write()` outputs a Unicode character as an integer between 1 and 65535.

Many of the subclasses of `Reader` and `Writer` have constructors which accept instances of `InputStream` and `OutputStream` as parameters.



`public abstract class Reader extends Object`

`Reader` is an abstract class for reading character streams. The only methods that a subclass must implement are `read(char[], int, int)` and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

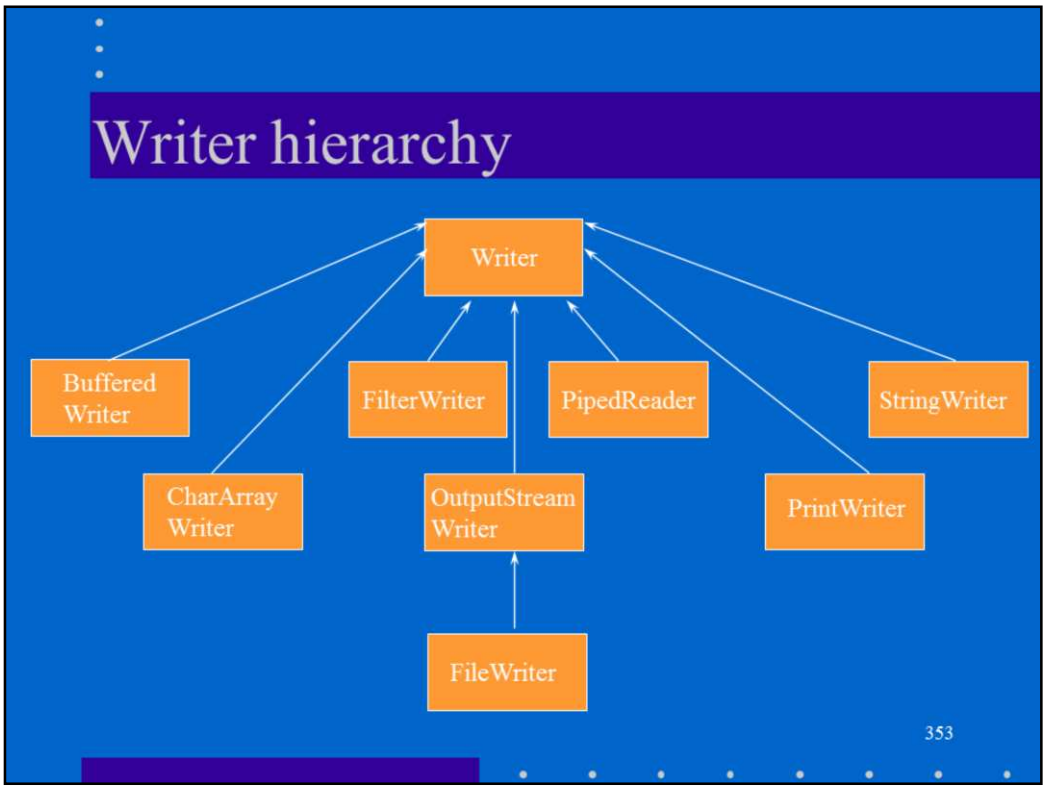
`public class InputStreamReader extends Reader`

An `InputStreamReader` is a bridge from byte streams to character streams. It reads bytes and translates them into characters according to a specified character encoding. The encoding that it uses may be specified by name, or the platform's default encoding may be accepted.

Each invocation of one of an `InputStreamReader's read()` methods may cause one or more bytes to be read from the underlying byte-input stream. For efficiency, consider wrapping an `InputStreamReader` within a `BufferedReader`; for example,

```
BufferedReader in =
          new BufferedReader(new
                    InputStreamReader(System.in));
```

Full details of these classes are in the online Java documentation.



`public abstract class Writer extends Object`

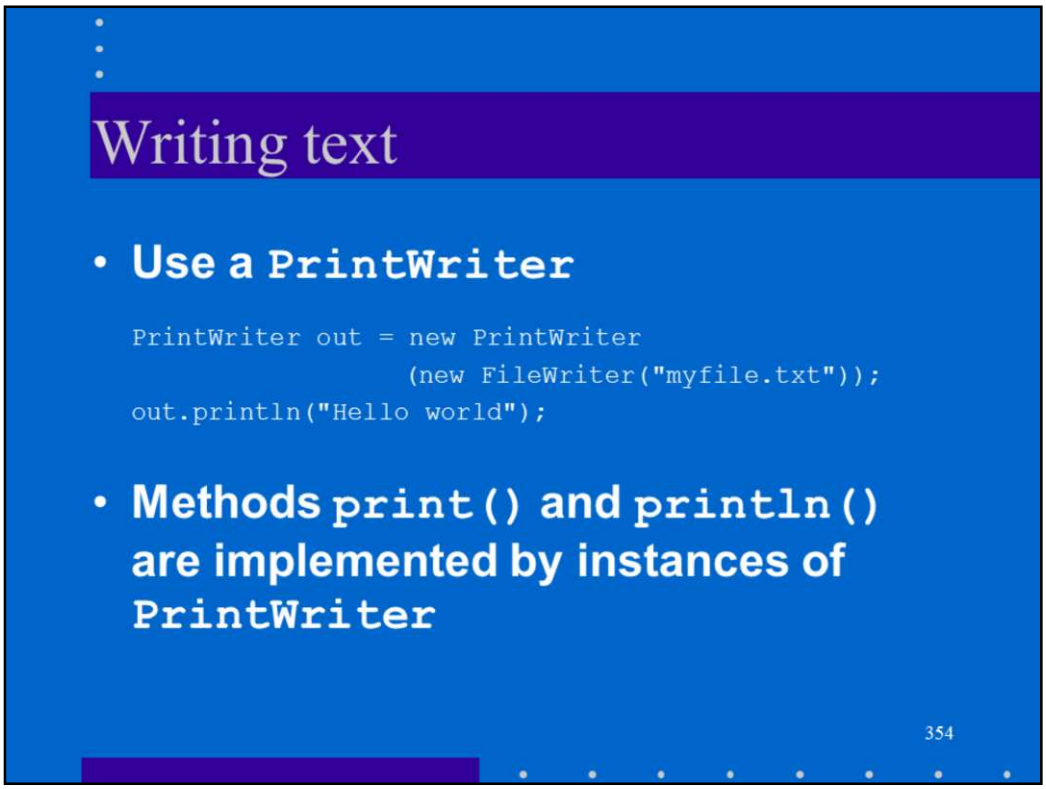
`Writer` is an abstract class for writing to character streams. The only methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

`public class PrintWriter extends Writer`

This class outputs print formatted representations of objects to a text-output stream. This class implements all of the print methods found in `PrintStream`. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println()` methods is invoked, rather than whenever a newline character happens to be output. The `println()` methods use the platform's own notion of line separator rather than the newline character.

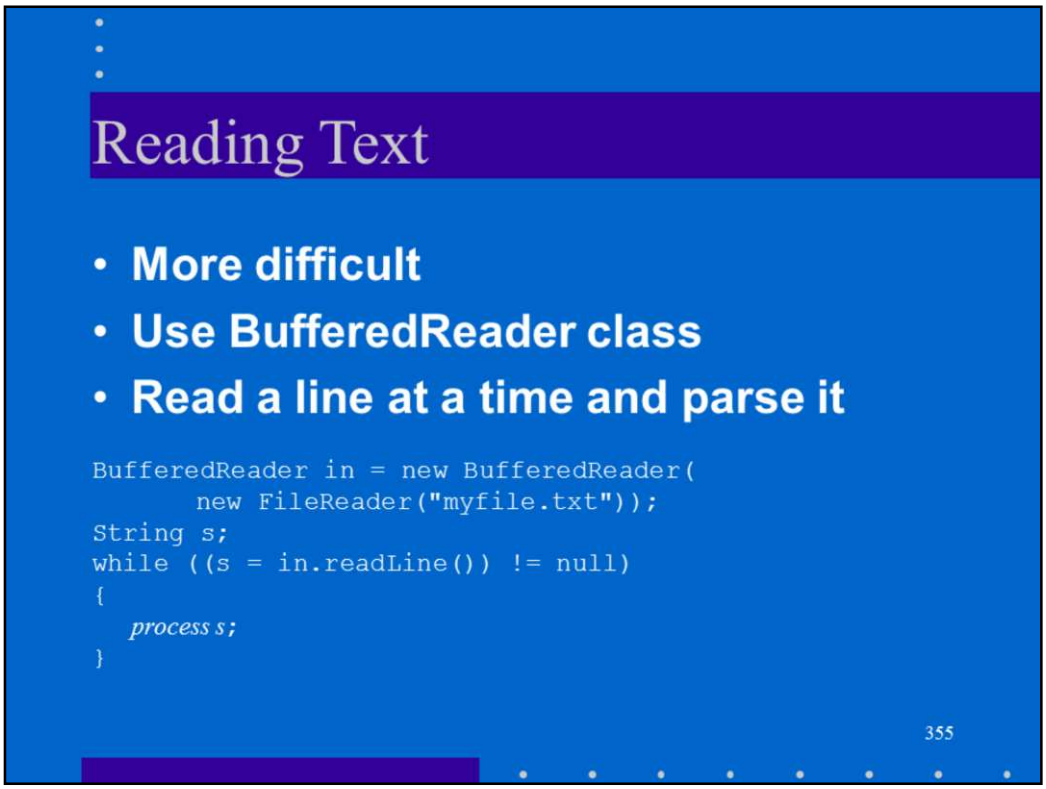
Full details of all the classes in this hierarchy are in the Java online documentation.



Internally, Java uses Unicode to represent characters. It is likely that Unicode will grow in popularity as a standard. At the moment, however, most operating systems encode text files as standard ASCII. If Java is to be used for producing files which can be read by applications such as Notepad, vi or textedit, then it must be able to output files encoded using ASCII.

In addition, it must cope with the different representation of text files on different hardware/software platforms. Under Windows a line in a text file is terminated with a carriage return character followed by a linefeed character. Under Unix a linefeed character terminates a text line. On a Macintosh a carriage return character terminates each line. The output classes must produce a text file appropriate to the platform on which they are executing.

The code above shows how an instance of the `FileWriter` class is used to open a stream on the file myfile.txt, and how this is used to create an instance of the `PrintWriter` class. The `PrintWriter` class implements the methods `print()` and `println()` that have been used extensively in the examples in these lectures.



Reading text in Java is trickier than writing out text, largely because the standard API does not come with classes that are very helpful. This is largely why this course has used Horstmann and Cornell's corejava classes for input.

The slide gives a clue as to how the corejava `Console` class is constructed. An instance of the `FileReader` class is opened on a file. This is used to create a instance of the `BufferedReader` class. `BufferedReader` implements a method `readline()`.

```
public String readLine() throws IOException
```

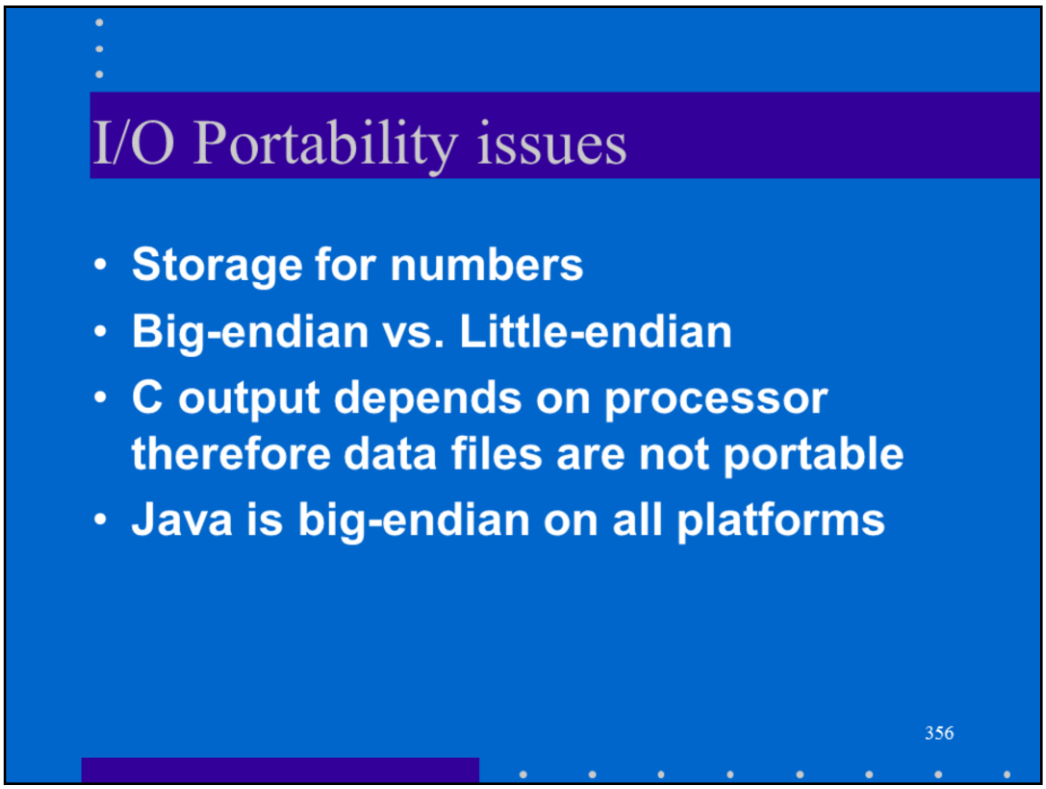
Read a line of text. A line is considered to be terminated by any one of a line feed ('`\n`'), a carriage return ('`\r`'), or a carriage return followed immediately by a linefeed. Returns a `String` containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached.

Once this string has been read it must be broken down into its constituent parts.

Alternatively, the stream may be read character by character using the `read()` method.

```
public int read() throws IOException
```

Read a single character.

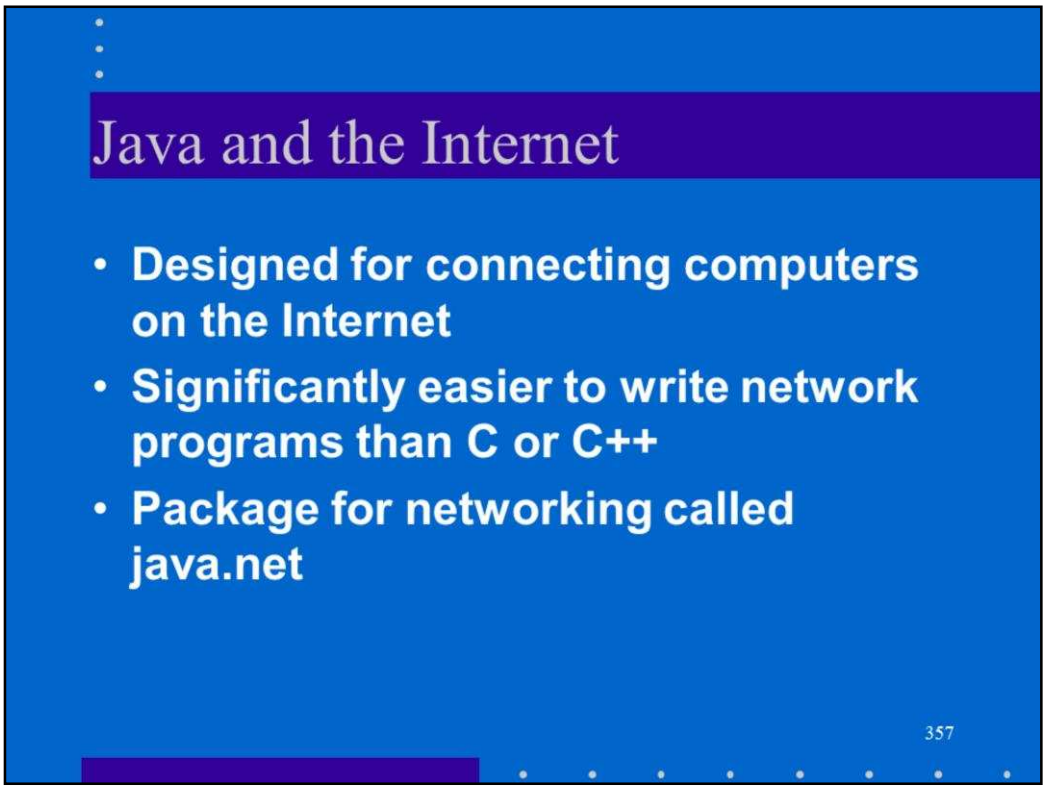

I/O Portability issues
• Storage for numbers
• Big-endian vs. Little-endian
• C output depends on processor therefore data files are not portable
• Java is big-endian on all platforms
356

Surprisingly, there is a portability issue involved with data files. There are two common techniques for storing numeric variables such as `int` or `float` in main memory. The first technique stores the most significant byte (MSB) of the number in the first of the four bytes in memory. This is known as big-endian. An alternative approach is to store the MSB in the last of the four bytes in memory.

In C or C++, if the program is running on a big-endian processor it will output a big-endian data file. If, on the other hand, it is running on a little-endian processor it will output a little-endian data file. This might not matter except that whilst a SPARC chip (Sun) is big-endian, a Pentium chip (Intel) is little-endian. A data file produced by a C or C++ program under Windows cannot be read directly by a C or C++ program running on a Sun Workstation! (Note this only applies to binary files, text has other problems!).

To ensure portability, Java is big-endian on all platforms. This means that a Java program will be able to read binary output from a Java program regardless of the platform it is implemented on. Conversely, a Java program cannot correctly read a binary file produced by a C or C++ program running on a Pentium processor.

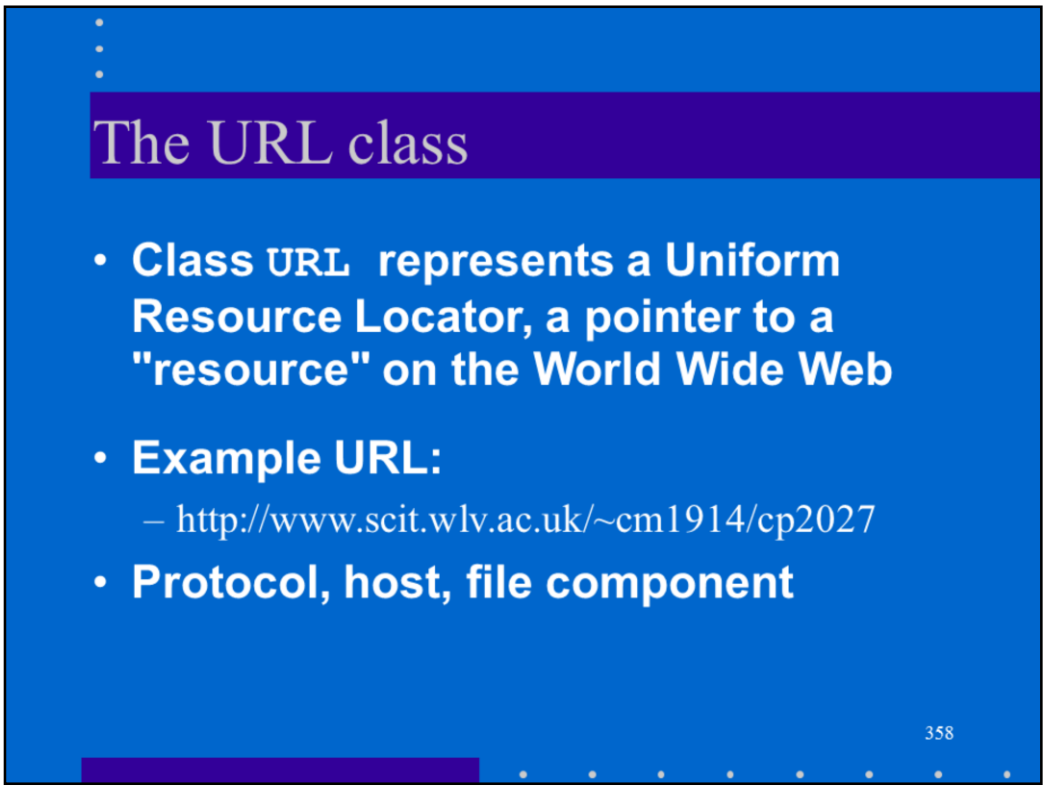
(NB: the terms big-endian and little-endian originated in Jonathon Swift's Gulliver's Travels).


Java and the Internet
• Designed for connecting computers on the Internet
• Significantly easier to write network programs than C or C++
• Package for networking called java.net
357

Java has been widely advertised as the language of the Internet. Part of the excitement about the language was its claims to make Internet programming easy.

Whilst Java uses the same techniques as C or C++ to talk to remote computers, it is undoubtedly true that its class libraries make it much simpler to construct Internet aware applications than these languages.

The package which contains the relevant classes is java.net.


The URL class
• Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web
• Example URL:
– http://www.scit.wlv.ac.uk/~cm1914/cp2027
• Protocol, host, file component
358

An example of a class which is specifically tailored to Internet access is the class `URL`. The term URL (Uniform Resource Locator) only came into existence with the emergence of the World Wide Web in the early 1990s. A URL is a mechanism for accessing a variety of Internet resources from any location on the Internet. Java implements URLs directly.

A URL consists of three components. For example:

http://www.scit.wlv.ac.uk/~cm1914/cp2027

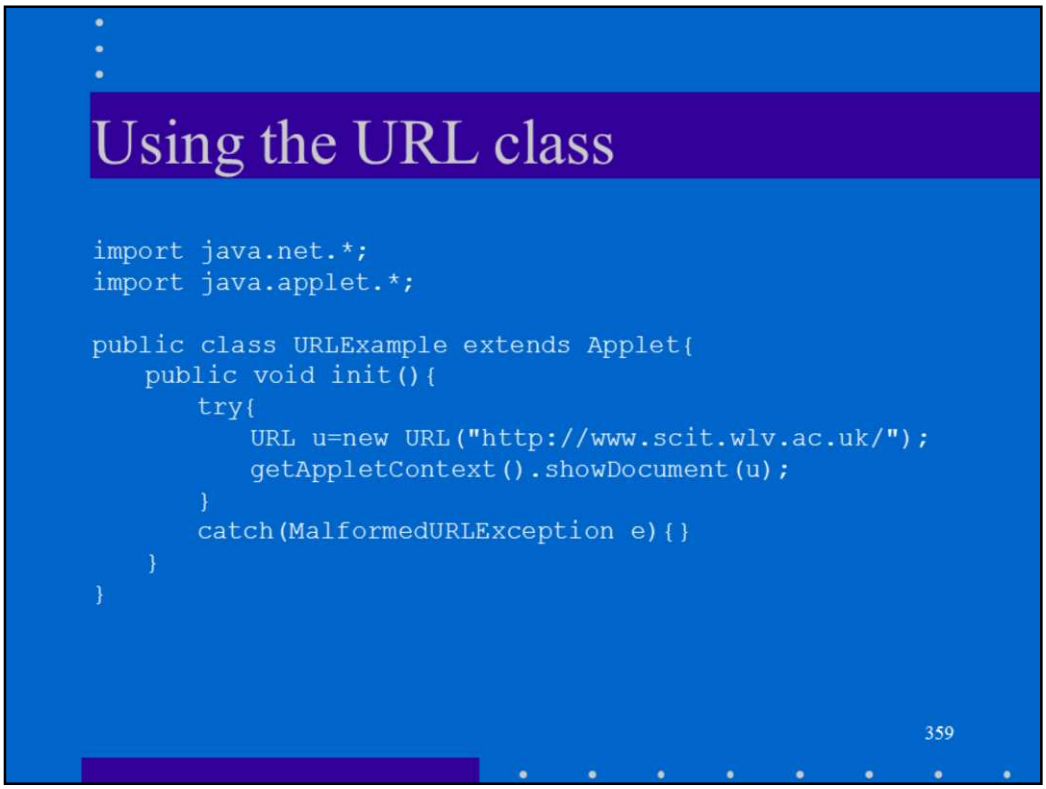
The first component http: identifies the protocol in use. In this case the protocol is http, HyperText Transfer Protocol, which is normally used to reference HTML pages. Alternative protocols include ftp and gopher.

The second component, www.scit.wlv.ac.uk, identifies the host on which the resource is located.

The third component instructs the source how to find the requested resource. In this case ~cm1914/cp2027 means: look in the home directory of user cm1914 for a directory called public_html, in that directory look for a subdirectory called cp2027, if there is a file called index.html, retrieve that, else retrieve the directory listing. The explanation of this component is implementation specific and may be realised differently on different platforms.

For a full description of URLs refer to

http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html



Here is an example of the use of class `URL`.

The applet imports the java.net package.

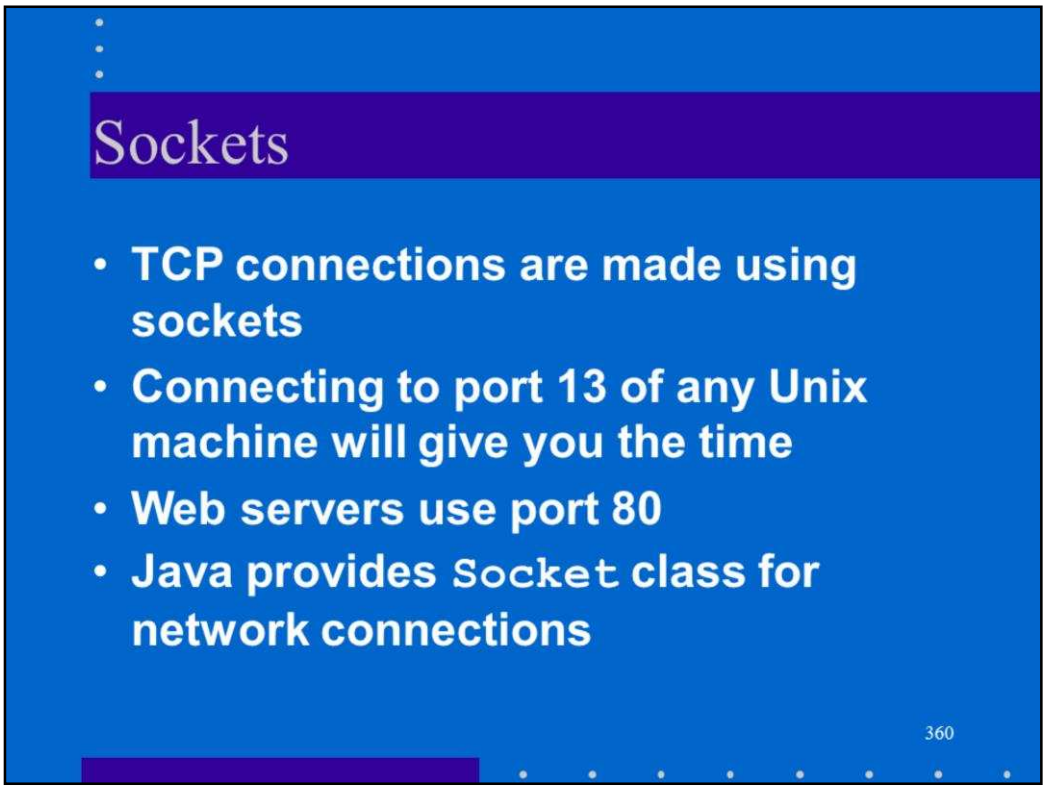
A `URL` is constructed from a `String`. This must be done within a `try..catch` block as the creation of a URL may generate a `MalformedURLException` and this must be caught. A compilation error will be flagged if this is not done.

In this case the catch block does nothing if an exception is generated.

Every applet executes in a browser. The browser is represented by an object of class AppletContext. Applets can obtain a reference to the object representing the browser in which they are executing by invoking the method `getAppletContext()`. An `AppletContext` has a number of methods associated with it, one of which is `showDocument()`. This method accepts an object of type `URL`, fetches the web page which it refers to and displays it in the browser.

The effect of this program is to replace itself within the browser with the SCIT server home page. It is an example of a badly behaved applet as it effectively disables the browser's BACK button. Going back results in a re-execution of the applet which promptly loads the home page again!

This applet will not work in the appletviewer as the appletviewer does not render HTML. It will work from a Java 1.1 enabled browser.


Sockets
• TCP connections are made using sockets
• Connecting to port 13 of any Unix machine will give you the time
• Web servers use port 80
• Java provides Socket class for network connections
360

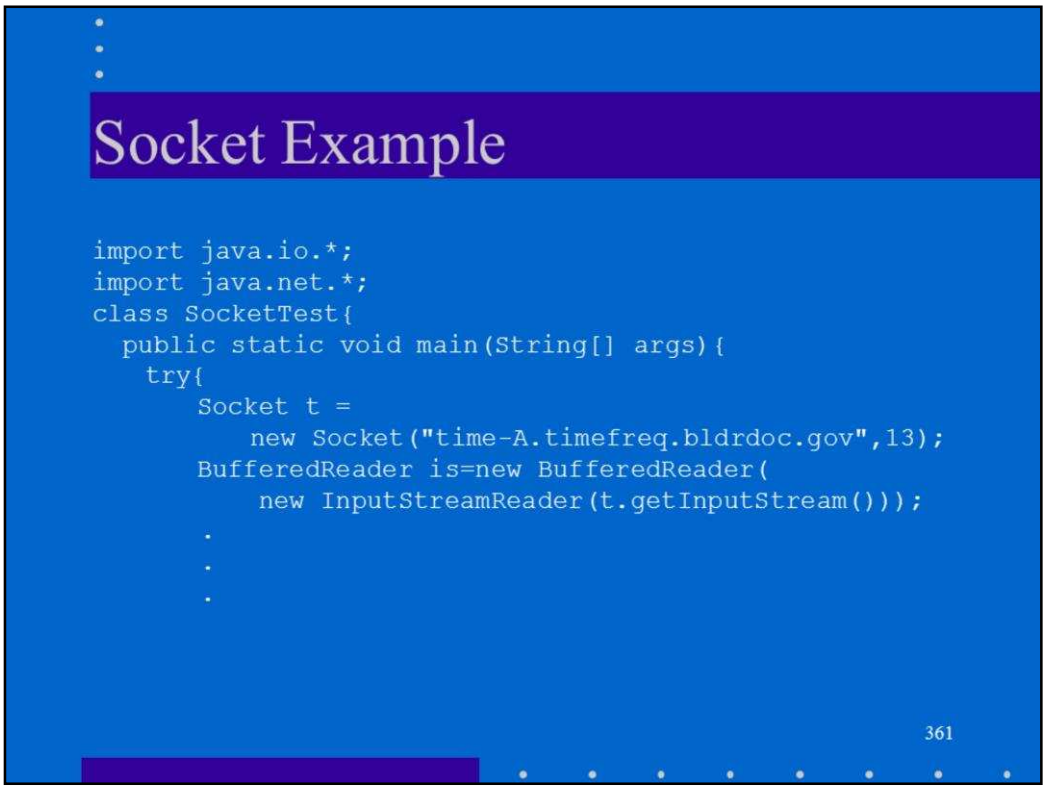
The generally accepted method of communicating across the Internet is via TCP/IP sockets. TCP/IP is the protocol used by the majority of Internet applications. Any computer which implements TCP/IP makes a number of ports available to the rest of the world. Remote applications may connect to these ports.

To make communication possible, a number of server programs are run which look out for programs trying to connect to particular ports, and respond to their requests.

Some ports are referred to as "well known ports". This means that by convention a particular type of server will be connected to that port. If you connect to port 13 of a Unix machine, you expect to find a server which will tell you the time in a string of characters. If you connect to port 80, you expect to find a web server. Not that these servers are only present if the system administrator has decided to run them. Most Unix machines, however, run the time server process (daemon).

This type of connection can be made in C or C++ via standard library calls. Java provides a class `Socket` to enable programs to use sockets connections.

Note that a program written in Java may quite happily use sockets to communicate with a program written a completely different language.



This program obtains the time from an atomic clock located at the National Institute of Standards and Technology in Boulder, Colorado. It is taken from coreJava 1.1 Vol. 2 written by Cay Horstmann and Gary Cornell, published by Prentice Hall, ISBN: 0-13-766965-8.

The program creates a connection to socket 13 to the Internet host with the domain name time-A.timefreq.bldrdoc.gov (the computer in Colorado). The `Socket` constructor may throw an exception so this has to be located in a `try…catch` block.

An instance of the `Socket` class has an associated input stream. A reference to this is obtained via the `getInputStream()` method. This is used to create a character stream which in turn is used to create a `BufferedReader`.

Socket Example continued

```
.
.
.
boolean more = true;
    while (more){
          String str = is.readLine();
          if (str==null) more = false;
          else System.out.println(str);
    }
 }
 catch(IOException e)
          {System.out.println("Error" + e);}
 }
}
```

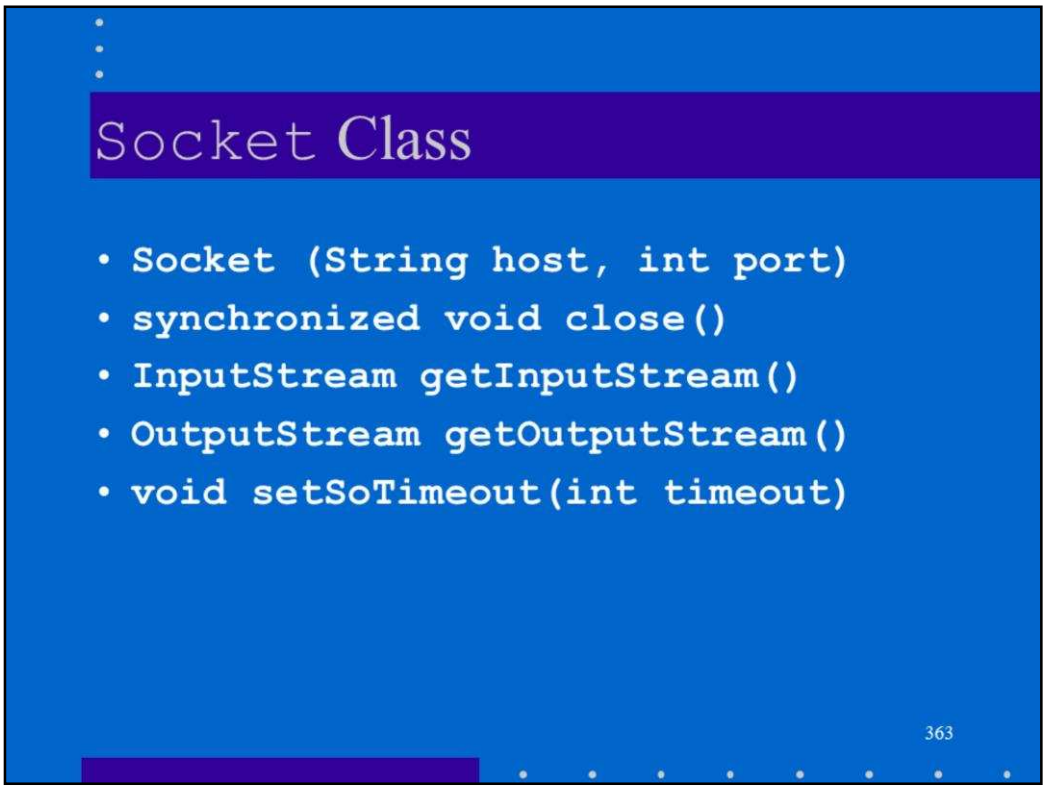
362

Once we have a `BufferedReader` we can use the `readLine()` method to obtain lines of text.

When a connection is made to port 13 on the machine in Colorado, the server on that port will simply issue the time as a string and close the port (finish the conversation).

This string will make its way across the Atlantic and finally end up in the input stream attached to the socket. The program reads from the stream and prints the string out.

This short program connects to another computer on the other side of the world, receives data from it and prints it out.



The constructor for the `Socket` class requires two parameters. The first, which is a string should contain the domain name for a host. The domain name of a host is its 'friendly' Internet address. Typical examples of domain names are "ccub.wlv.ac.uk" and "scitsc.wlv.ac.uk".

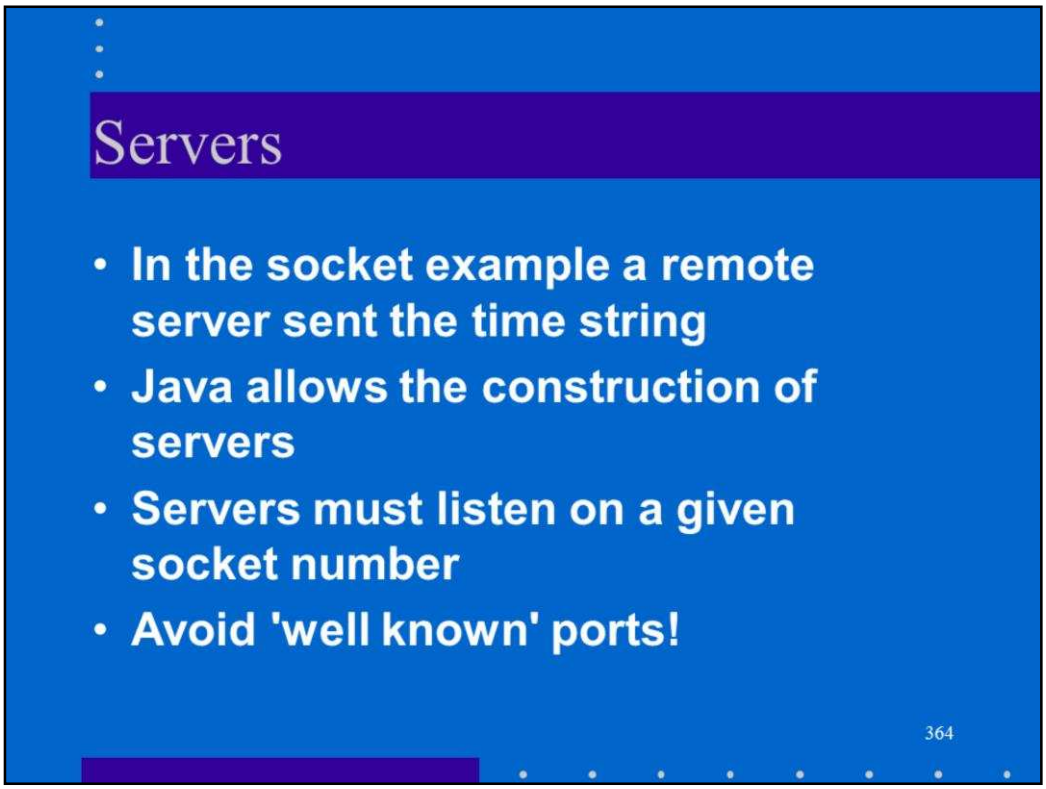
The `close()` method releases resources used by a socket.

`getInputStream()` returns an object of class `InputStream` which references the input stream associated with the socket.

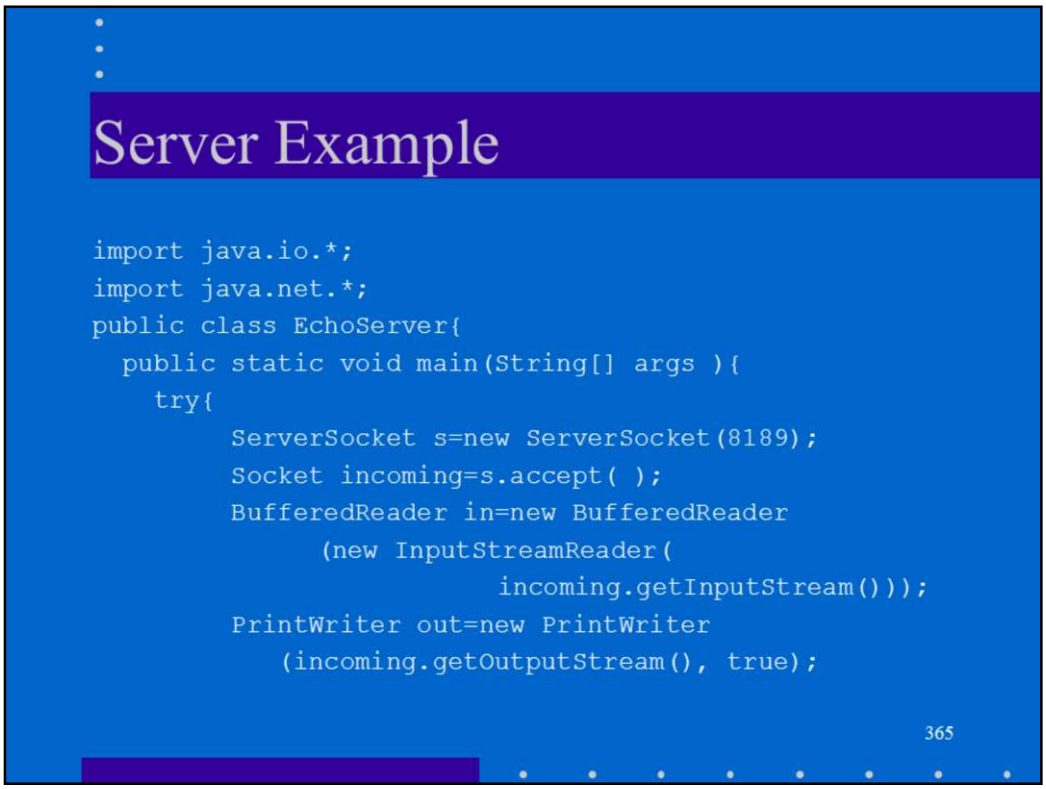
`getOutputStream()` returns an object of class `OutputStream` which references the output stream associated with the socket.

`setSoTimeout()` set the maximum time in milliseconds that the thread will be blocked for when it reads from the socket. At the end of this time an exception is thrown

A full list of the methods which apply to sockets are available in the Java online documentation


Servers
• In the socket example a remote server sent the time string
• Java allows the construction of servers
• Servers must listen on a given socket number
• Avoid 'well known' ports!
364

Connecting to a socket achieves something because, behind the port specified in the constructor, there is a server which handles the connection. Java allows the programmer to connect to a server and also to create a server. The most important feature of a server is that it listens on a particular port. The port number must be known to all the programs that wish to connect to it. You can choose any port number to listen on, but you should avoid those numbers already associated with standard services. You can access a list of well known ports at http://www.library.ucg.ie/CIE/RFC/1700/4.htm. On a multi-user machine avoiding a port number that is used on an ad hoc basis by another user can turn out to be harder!



Here is a program which implements a simple server. As with the previous example, it is taken from coreJava 1.1 Vol. 2 written by Cay Horstmann and Gary Cornell, published by Prentice Hall, ISBN: 0-13-766965-8.

The program creates a `ServerSocket` object and specifies that it will listen on port 8189. This must be done within a `try…catch` block as the constructor can throw an `IOException`.

The `accept()` method blocks the thread until someone tries to connect to the port. On unblocking it returns a reference to a `Socket` object.

An object of class `BufferedReader` is created to deal with input from the socket.

An object of class `PrintWriter` is created to direct output to the socket.

Server example continued

```
            out.println("Hello! Enter BYE to exit.");
            boolean done=false;
            while (!done){
               String str=in.readLine();
               if (str==null)done=true;
               else{
                 out.println("Echo: " + str);
                 if(str.trim().equals("BYE"))done=true;
               }
            }
            incoming.close();
         }
         catch (Exception e){System.out.println(e);}
      }
   }
```

366

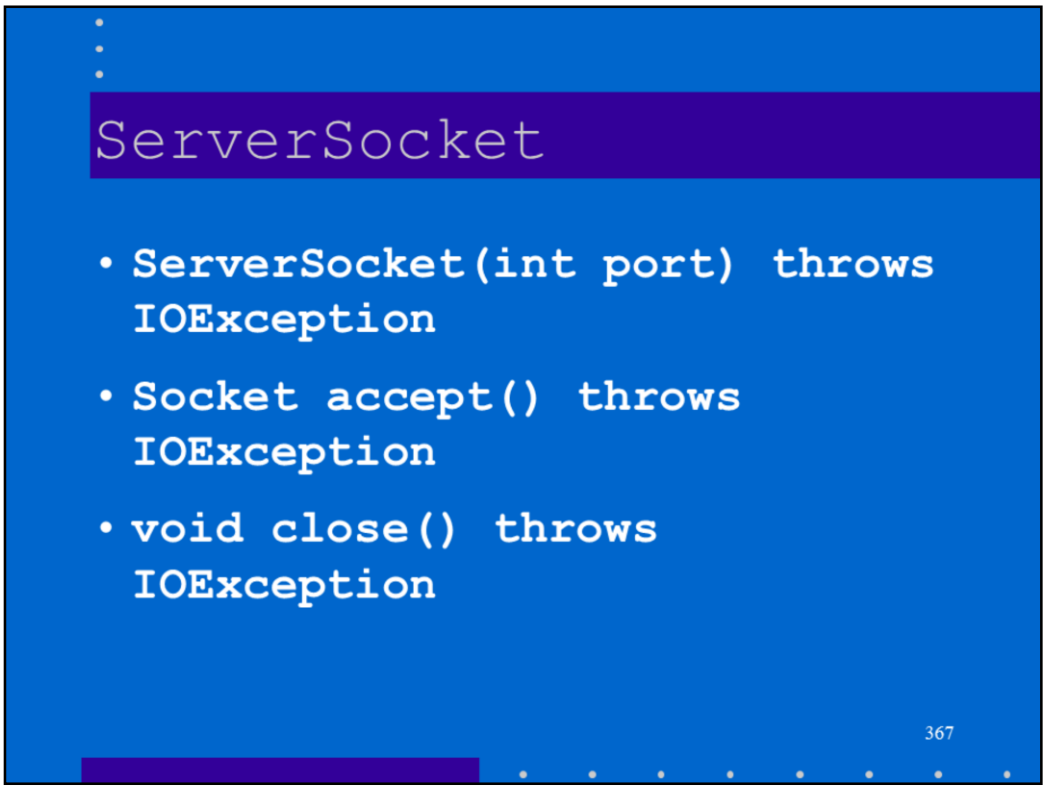
The output stream associated with the socket is sent the text “Hello! Enter Bye to exit”. This text will be sent to the input stream of the remote program which connects to the port.

The program loops reading from the input stream associated with the socket until it reads the string “BYE”. Each time it reads from the socket it sends the input back to the sender.

When it receives the string “BYE” it closes the socket and exits.

The program could be used in the following way. It could be set running (on a PC or a Unix server it doesn’t matter). Someone on the other side of the world, who was in the know, could connect using the Internet address of the machine the server was running on and the port number.

The first thing anyone who connected would receive would be the string “Hello……”. Following this, any text they entered would be echoed back to them, until they entered “BYE”, when the connection would be dropped.



`public ServerSocket(int port) throws IOException`

Creates a server socket on a specified port. A port of 0 creates a socket on any free port.

The maximum queue length for incoming connection indications (a request to connect) is set to 50. If a connection indication arrives when the queue is full, the connection is refused.

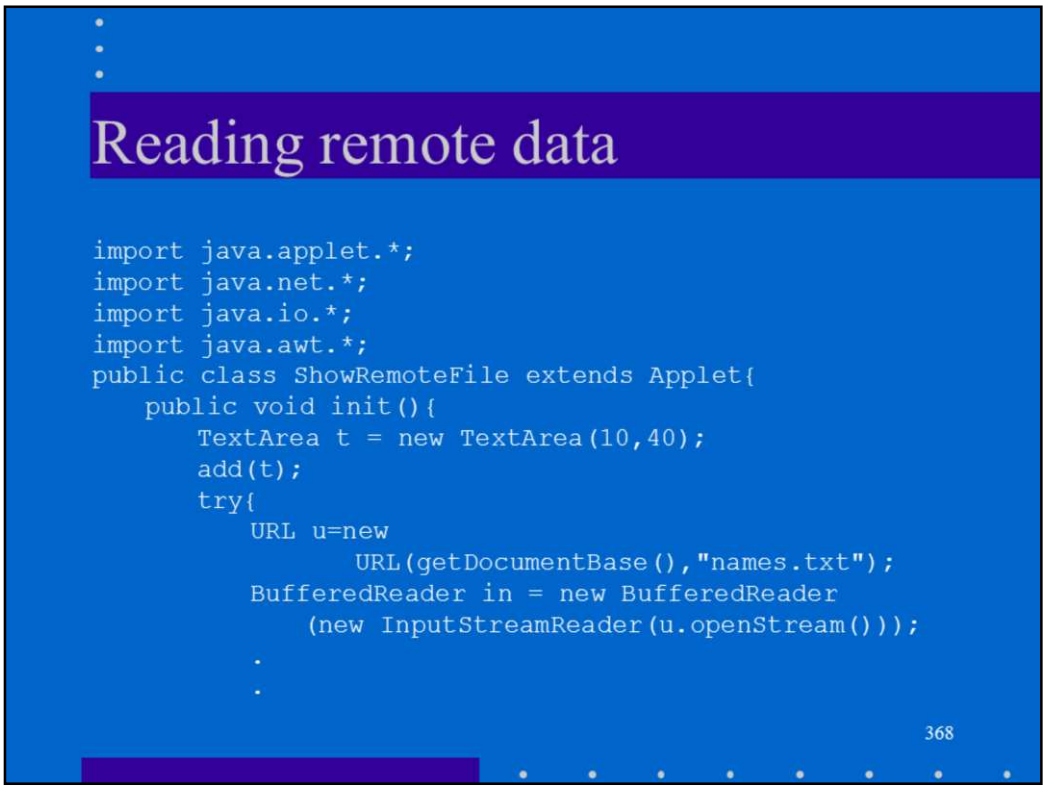
`public Socket accept() throws IOException`

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

`public void close() throws IOException`

Closes this socket.

Further methods appear in the Java online documentation.



This code shows how an applet can read the contents of a file on the machine from which it was downloaded (often an entirely different machine from the one on which the applet executes).

The applet creates and displays a `TextArea` object.

It then creates a `URL` object from the directory in which the downloaded HTML file was found (as returned by `getDocumentBase()`) which references a file called names.txt. In this case names.txt must be located in the same directory as the HTML file which referred to the applet.

`URL` objects implement a method `openStream()` which returns an `InputStream`. This is used to create an object of class `BufferedReader`.

Reading remote data (continued)

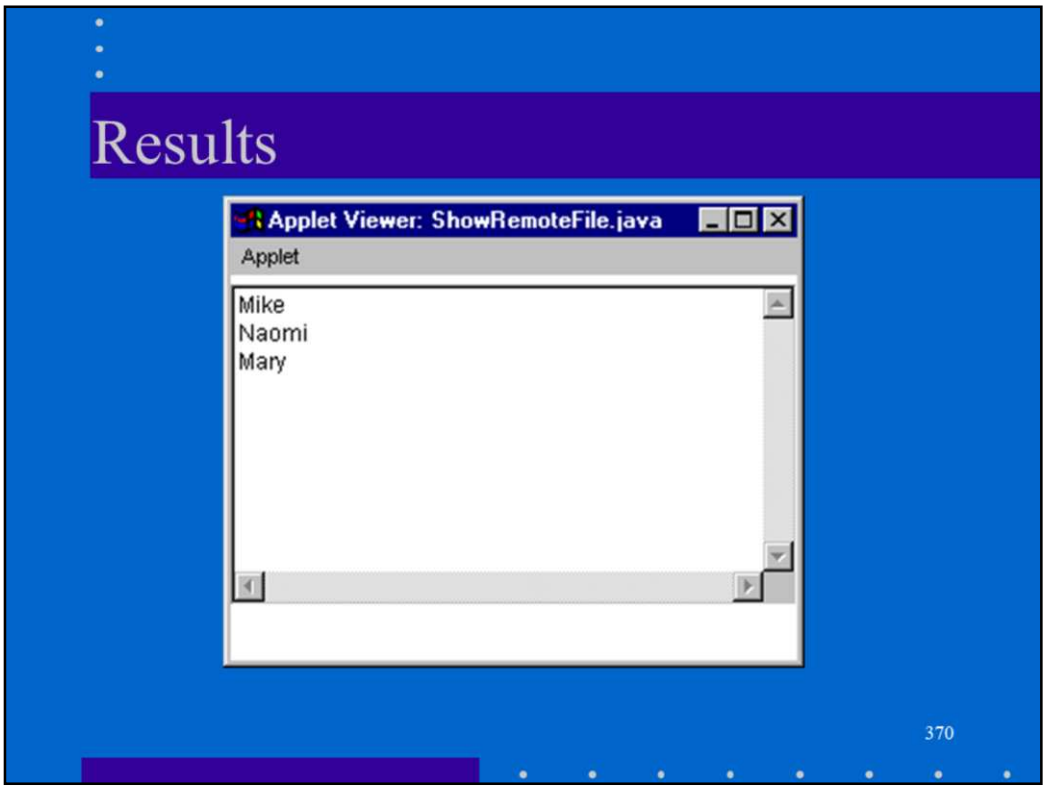
```
            .
            boolean done = false;
            while(!done){
                String str=in.readLine();
                if(str==null)done=true;
                else{
                    t.append(str);
                    t.append("\n");
                }
            }
            in.close();
        }
        catch (Exception e){}
    }
}
```

369

The remote connection can now be read as though it were a local input stream. The program loops round until a `null` indicates that the end of the stream (end of remote file) has been reached. It places each line of data into the `TextArea`.

On reading a `null` the program closes the connection and exits the initialisation method.

Again, this technique could be used to make data available to any Internet connected computer in the world.

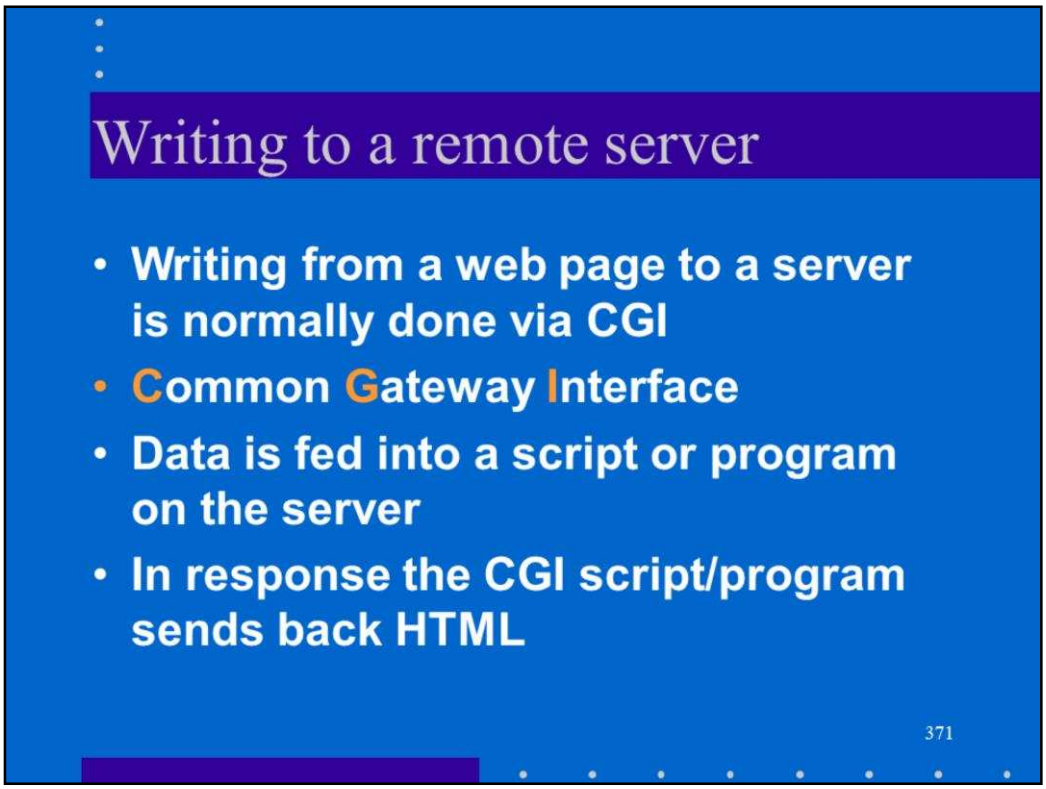


This is the result of executing the applet shown on the previous two slides. In this case the file names.txt contained the three lines:

Mike

Naomi

Mary

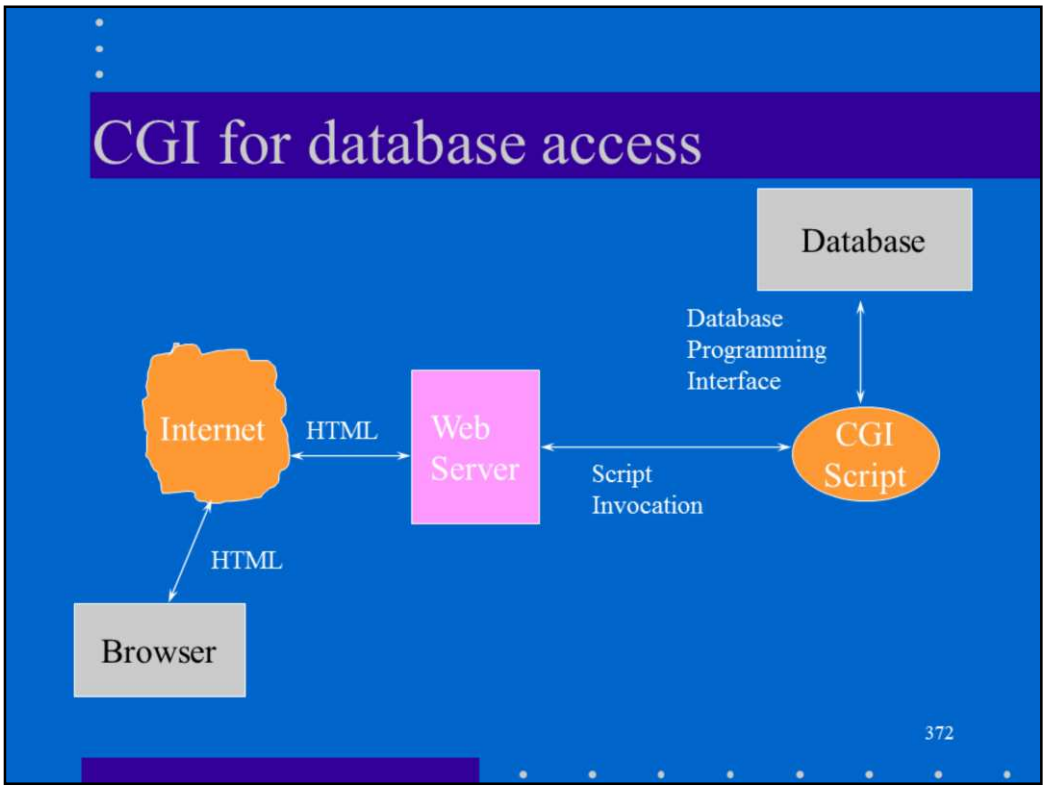

Writing to a remote server
• Writing from a web page to a server is normally done via CGI
• Common Gateway Interface
• Data is fed into a script or program on the server
• In response the CGI script/program sends back HTML
371

The previous example showed how to read a file located on a web server. It is possible to write to a file located on a web server. It is possible to do this without using Java, and the normal method is through CGI which is an abbreviation for Common Gateway Interface. CGI simply specifies how a browser can initiate and pass parameters to a script or program running on a server.

A link on the HTML page specifies what routine should be run if the link is selected and may also pass parameters, which have been entered into a form, created via HTML, and displayed in the browser.

The CGI script or program is invoked by the web server and passed the parameters from the browser. It generates output in the form of HTML. This HTML is passed backed to the browser which displays the results.

This is illustrated on the slide on the next page.



This illustrates the process of accessing a remote database using a CGI script.

The user fills in a form on the browser.

This generates a request which specifies a CGI script or program. The request is passed to the web server.

The web server invokes the script or program.

The script or program interprets the parameters and constructs a database query. The query is applied to the database and the results are incorporated into a page of HTML.

The HTML is passed to the web server which forwards it across the Internet to the browser.

The browser renders the HTML and the user sees the results on the screen.

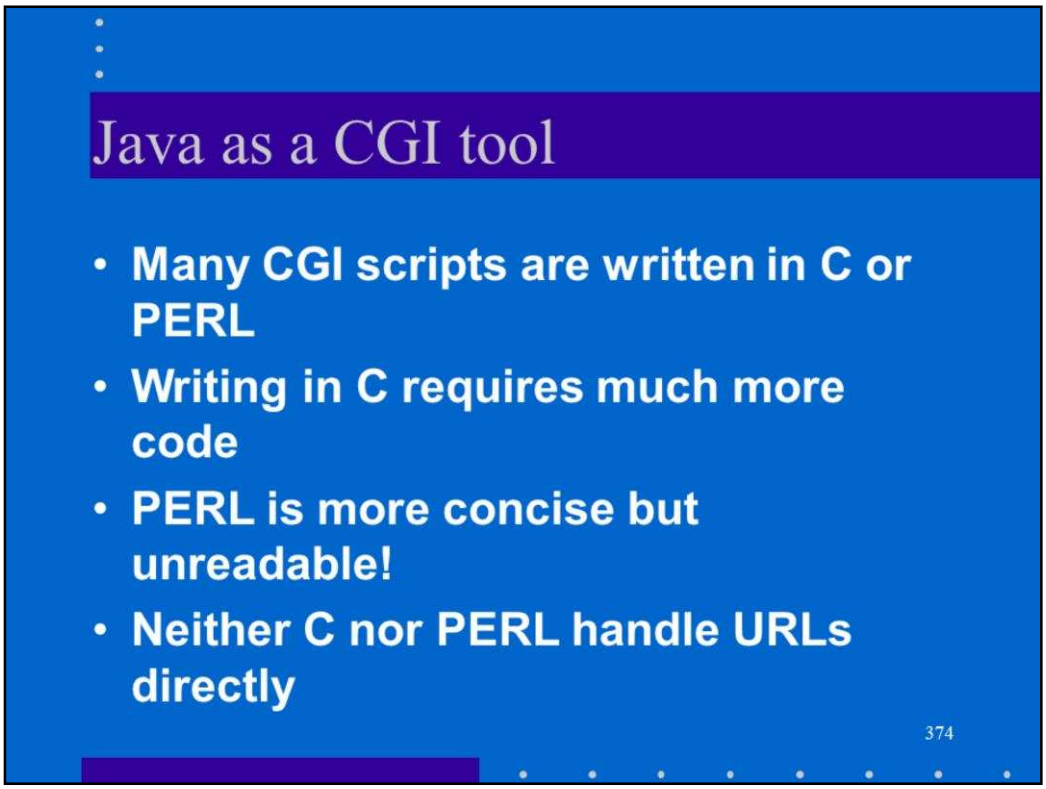
The CGI script or program can read or write to the database, so the user could use the HTML page as a data entry screen.

Simple Java CGI

```
public class HelloWorld{
    public static void main(String[] args){
        System.out.println
                    ("Content-type: text/html\n\n");
        System.out.println("<html>");
        System.out.println
            ("<head><title>Hello World</title></head>");
        System.out.println("Hello World");
        System.out.println("</body>");
        System.out.println("</html>");
    }
}
```

373

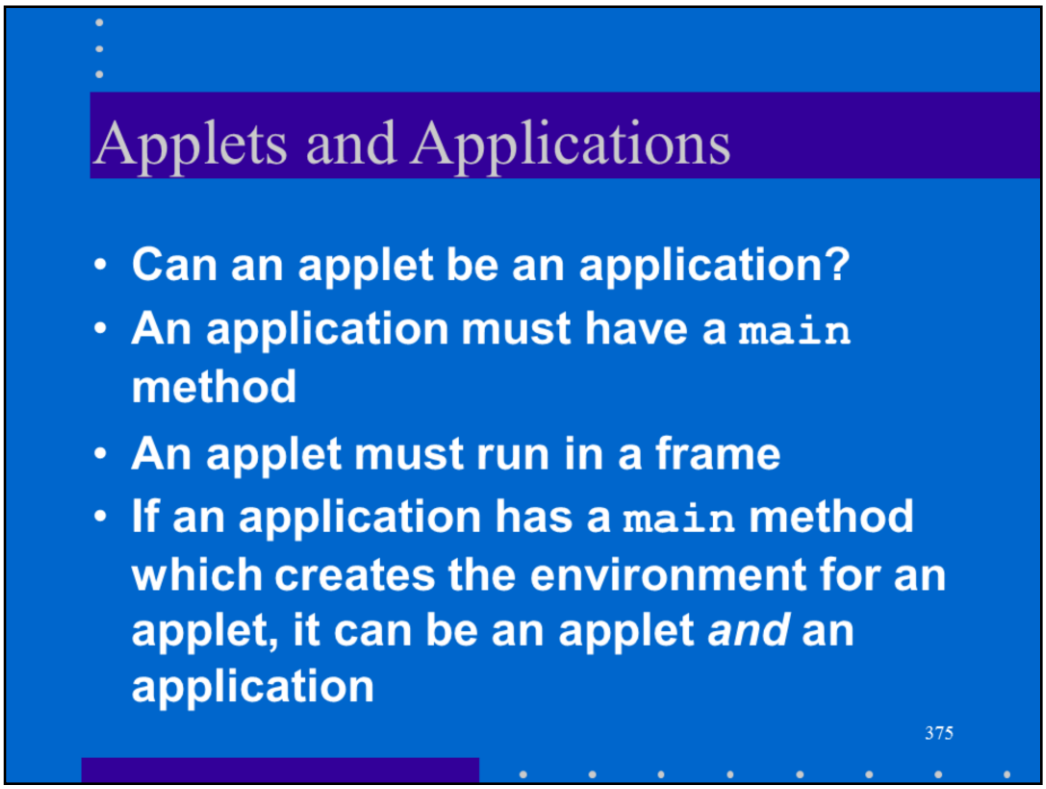
Here is a simple CGI program written in Java. If invoked by a user selecting a link in a browser, it responds by creating a simple piece of HTML which displays the text “Hello World”. You can see that the output includes all the necessary HTML tags.


Java as a CGI tool
• Many CGI scripts are written in C or PERL
• Writing in C requires much more code
• PERL is more concise but unreadable!
• Neither C nor PERL handle URLs directly
374

You can write a CGI script in almost any language. C, PERL (an extremely powerful, but very odd, interpreted language) and Unix shell scripts are often favourites. On Windows NT servers CGI scripts are often written in Visual Basic. PERL and C are by far the most common. Compared to Java, CGI scripts written in C often require many more lines of source code. Scripts in PERL can be shorter, however PERL has been descried as a 'write only' language, i.e. once written the scripts tend to be unmaintainable. Only Java has a built-in capacity to deal with URLs.

Java is therefore a good candidate for using with CGI, however a requirement is that the server can recognise a reference to a Java routine, and load the JVM in order to execute it.

Java can be used to invoke a CGI script by connecting directly to a web server (using a socket) and passing appropriate HTML text. The web server would not be able to tell the difference between an invocation of a script from a Java program and the invocation of a script from a browser.


Applets and Applications
• Can an applet be an application?
• An application must have a main method
• An applet must run in a frame
• If an application has a main method which creates the environment for an applet, it can be an applet and an application
375

This course has drawn a distinction between applets and applications. Is it possible to write code which is simultaneously an application and an applet. This would have the advantage of being executable locally from the command line and remotely in a browser.

Applications must have a method called `main()` which specifies the code which will be executed when the application is invoked.

Applets run in a browser. The browser provides the windowing environment in which the applet executes.

If the `main()` method of an application creates an environment similar to that supplied by a browser then an applet can be run within it.

AppletFrame class

```
public class AppletFrame extends Frame{
   AppletFrame(Applet a, int x, int y){
        setTitle(a.getClass().getName());
        setSize(x,y);
        add("Center",a);
        a.init();
        show();
        a.start()
   }
   ……..
}
```

376

Here we have a class which can create the necessary environment for an applet. The key aspect is that it is a window and extends the `Frame` class. (It might be more sensible to extend a subclass of `Frame` which responds to the `CloseWindow` event!).

The constructor for the class accepts three parameters: an `Applet` object and two integers which specify the width and height of the window the applet requires.

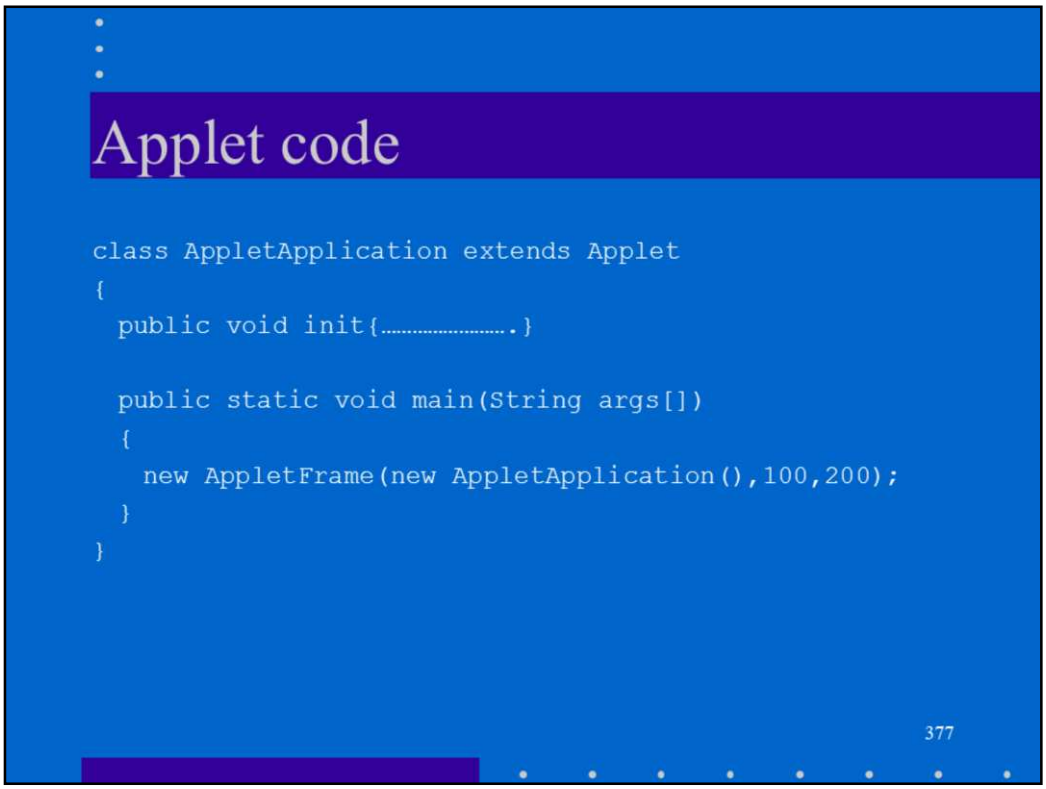
The name of the applet is retrieved and used to set the title of the window.

The width and height parameters are used to set the size of the window.

The `init()` method of the applet is invoked and the window is displayed.

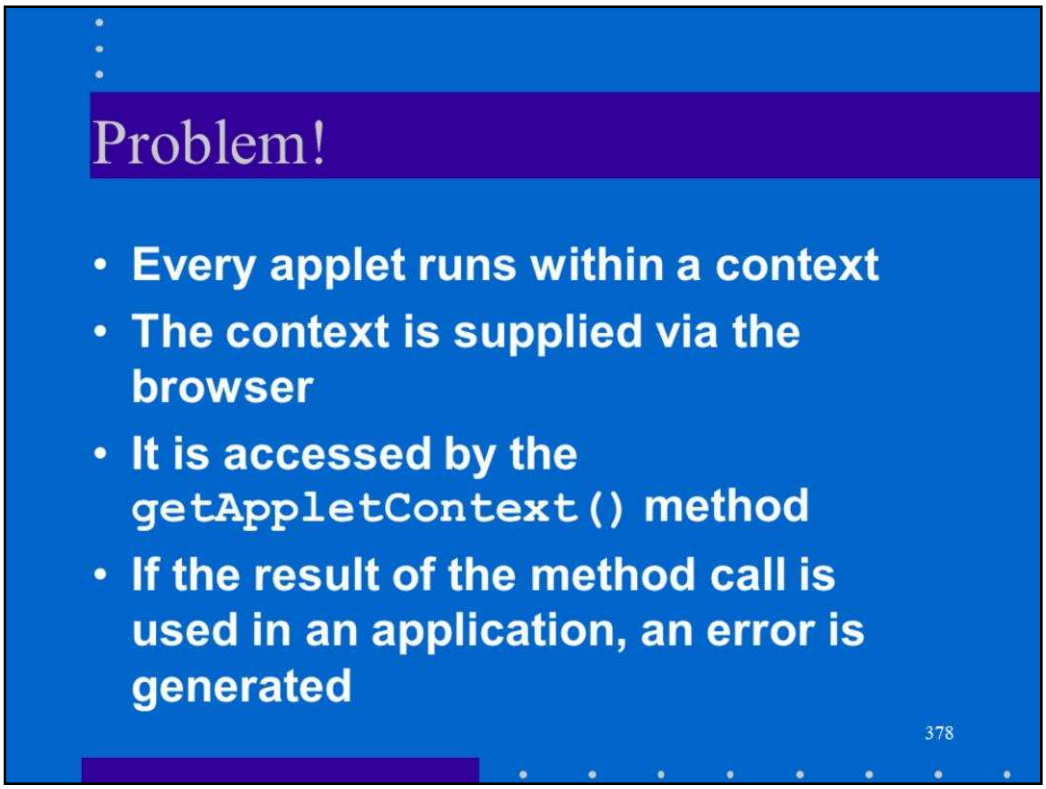
Finally the applet's `start()` method is called.

This will run the applet within a window.

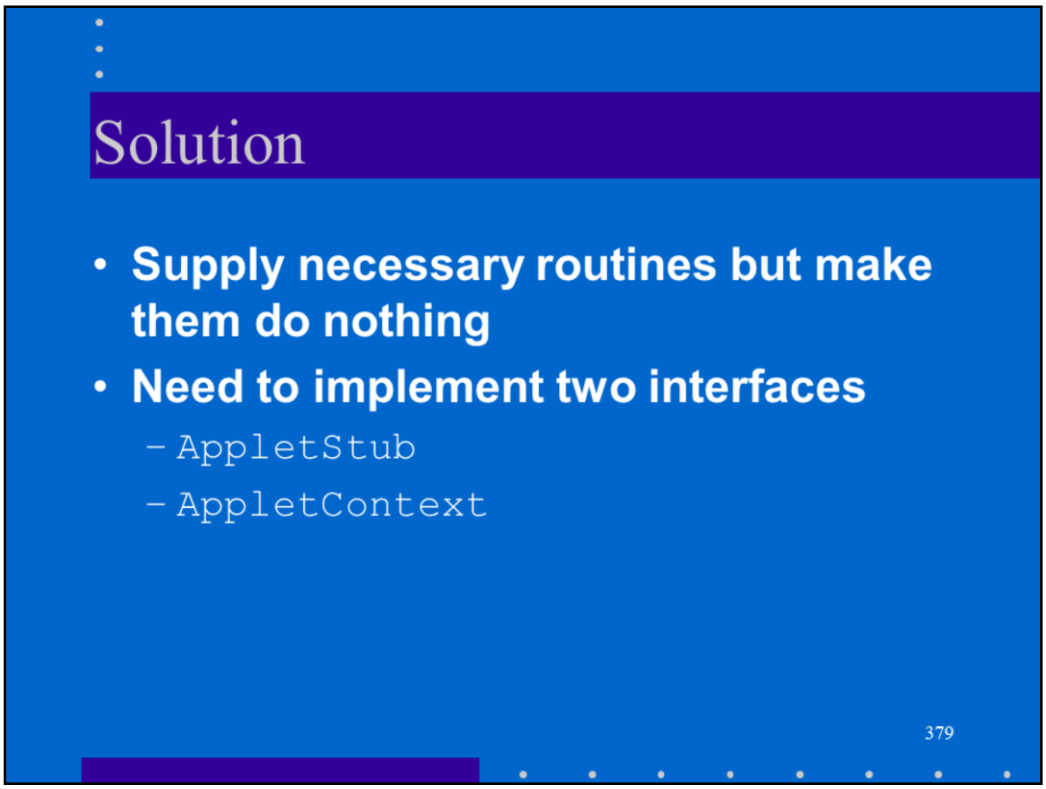


It is now possible to define a class which is a subclass of `Applet` but also contains a `main()` method. The `main()` method will only be invoked if the applet is run as though it were an application. It will create a `AppletApplication` object and this will execute the applet in a window.

If the applet is executed in a browser it will behave normally and `main()` will not be invoked.


Problem!
• Every applet runs within a context
• The context is supplied via the browser
• It is accessed by the getAppletContext() method
• If the result of the method call is used in an application, an error is generated
378

There is a problem with this solution. An applet can ask questions about the browser it is running in. It does this by calling the methods of its associated `AppletContext`. If the applet is run in an ordinary window then there isn't an associated `AppletContext` and, if the applet needs to find out about the browser's properties, an error will occur.

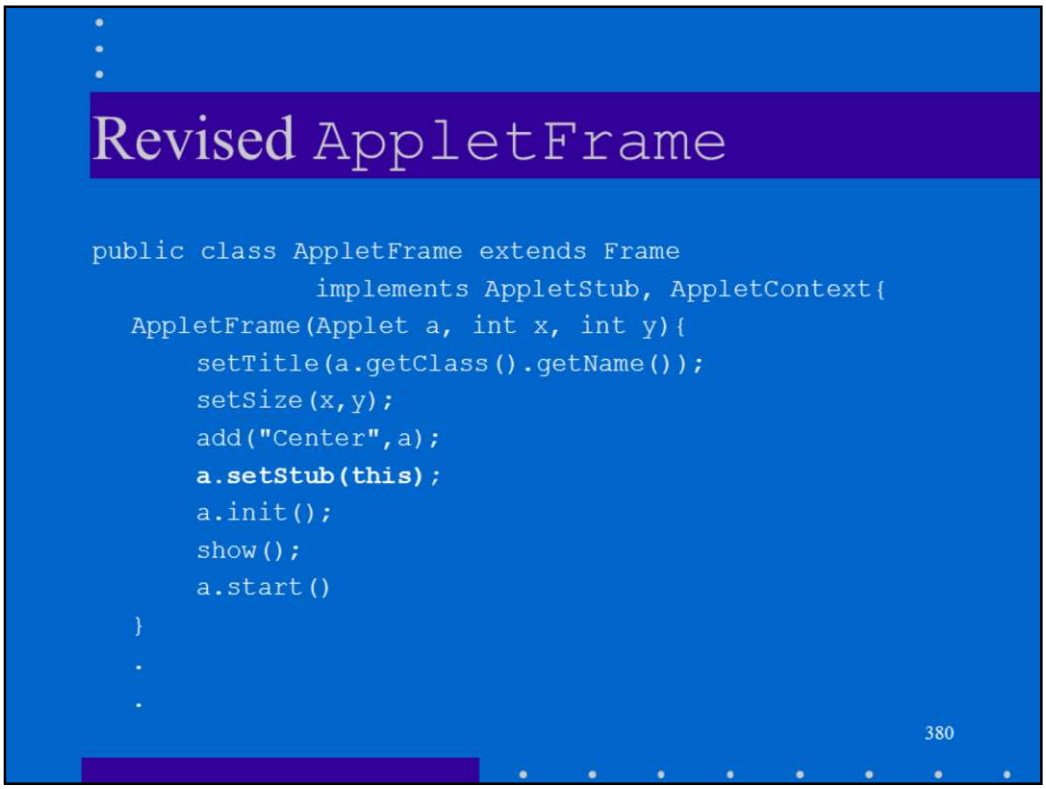


The problem occurs because the applet is trying to call routines which are implemented by a browser, but not by a frame.

One solution is to add all the browser capabilities to the application.

A simpler solution is just to implement all the necessary methods but to make them “do nothing” methods.

The methods which must be implemented are contained in the interfaces `AppletStub` and `AppletContext`. This affects class `AppletFrame`.



The class is altered so that it states that it implements the two required interfaces.

One extra line is added which is a call to `setstub()`. This method sets up the `AppletFrame` class as the context for the applet. A compilation error would be generated if the parameter to `setStub()` did not reference an object which implemented the `AppletContext` and `AppletStub` interfaces. In this example, it is clear that `this` fulfils this criteria.

Revised AppletFrame (cont)

```
    public boolean isActive() {return true;}
    public URL getDocumentBase() {return null;}
    public URL getCodeBase() {return null;}
    public String getParameter(String name){return "";}
    public AppletContext getAppletContext()
                                        {return this;}
    public void appletResize(int width, int height){}
    public AudioClip getudioClip(URL u){return null;}
    public Image getImage(URL u){return null;}
    public Applet getApplet(String name){return null);
    public Enumeration getApplets(){return null;}
    public void showDocument(URL u){}
    public void showDocument(URL u,String s){}
    public void showStatus(String status){}
}
```

381

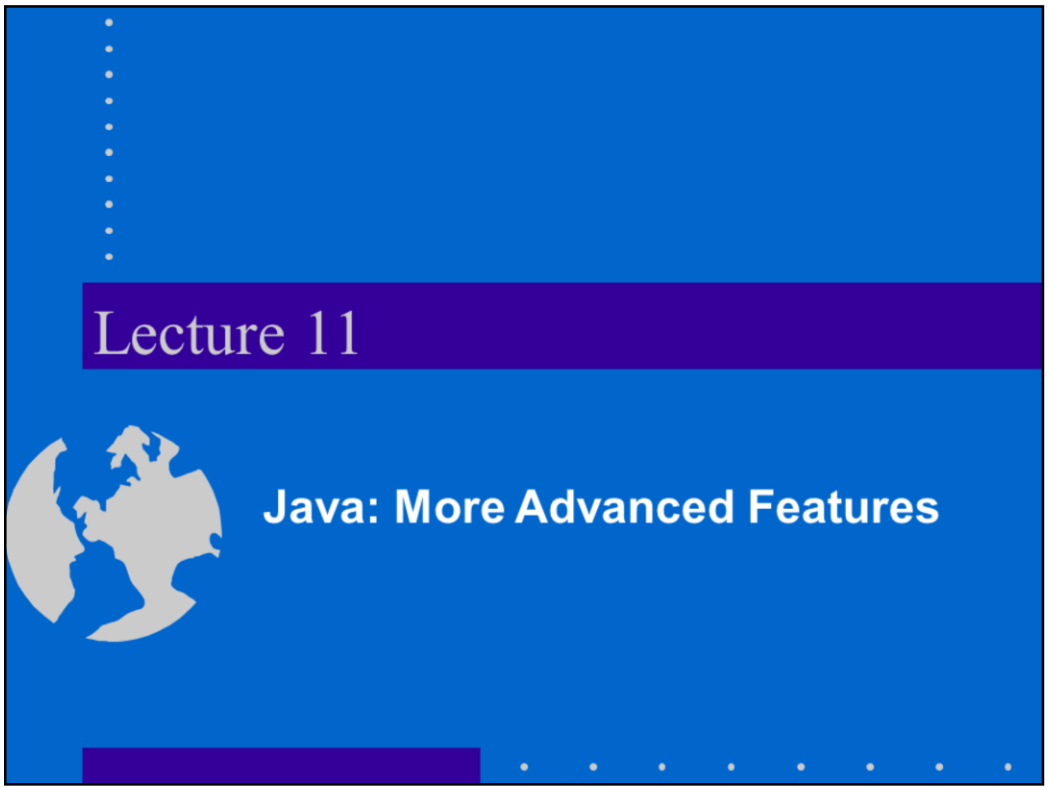
The class must, of course, contain code for all the methods in the two interfaces. Here are all the "do nothing" methods which are required.

With these classes it is possible to run an applet as though it were an application. Whilst no execution errors will occur, the applet may not behave the same as it would if it were run within a browser. This will occur because the applet is calling the "do nothing" methods as though they performed a useful task.

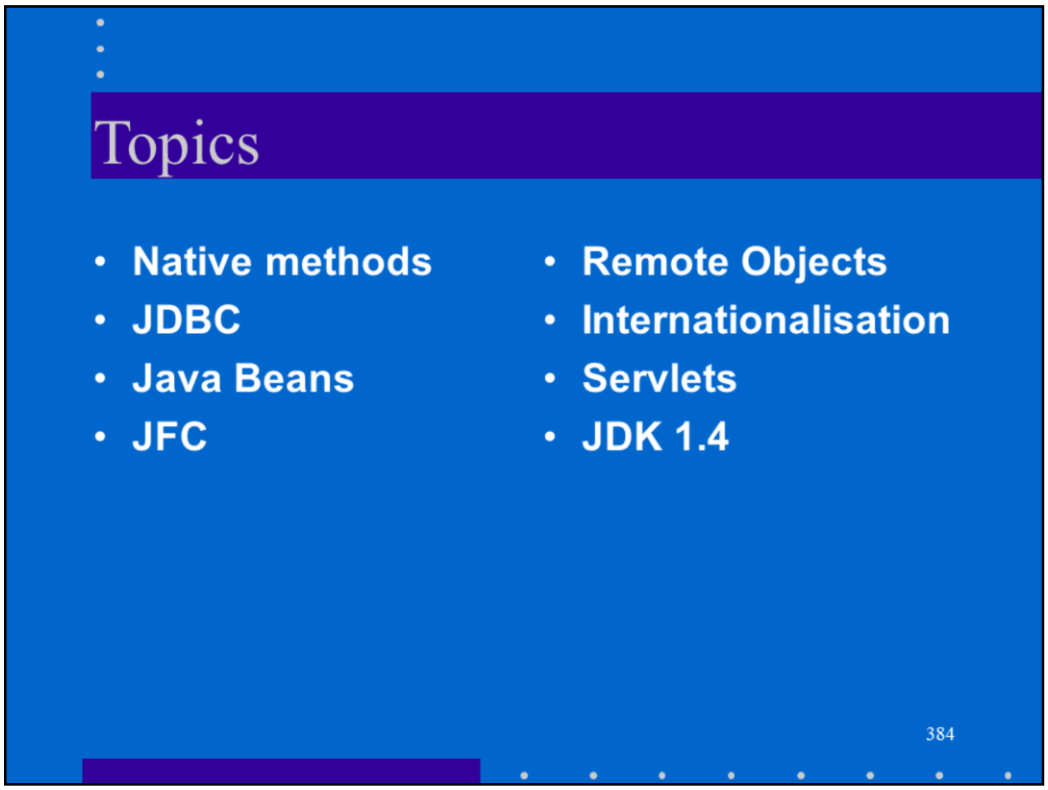
It is clear that the appletviewer program has been constructed in a manner similar to `AppletFrame`.

Summary

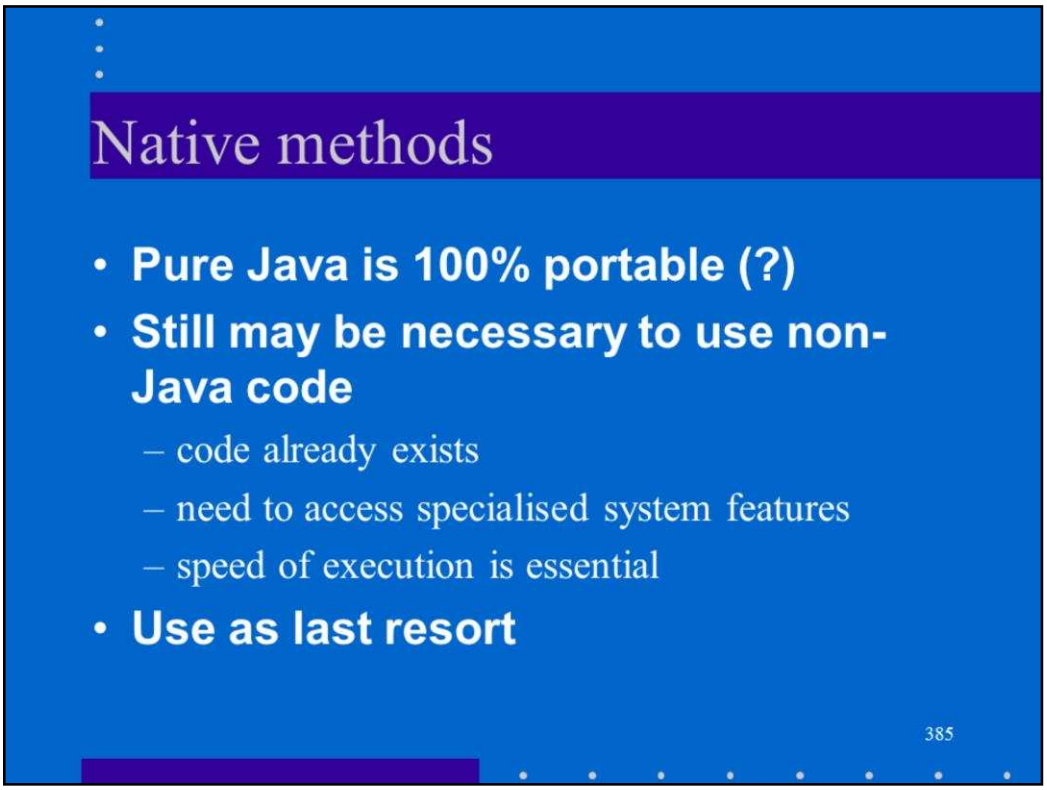
- Streams
- `InputStream`
- `OutputStream`
- `Reader`
- `Writer`
- `URL`
- `Socket`
- `ServerSocket`
- Reading from a remote server
- CGI scripts
- Running an applet within an application



This lecture presents a brief look at some of Java's advanced features.



What is presented in this lecture is a selection of Java developments. There are many more directions in which Java is being developed. For the latest trends visit Sun's Java site http://java.sun.com.


Native methods
• Pure Java is 100% portable (?)
• Still may be necessary to use non-Java code
– code already exists
– need to access specialised system features
– speed of execution is essential
• Use as last resort
385

The advantage of writing applications entirely in Java is that in theory these applications should be 100% portable. In practice we know that because of implementational difficulties this isn't yet the case. Still, it's a goal worth aiming at.

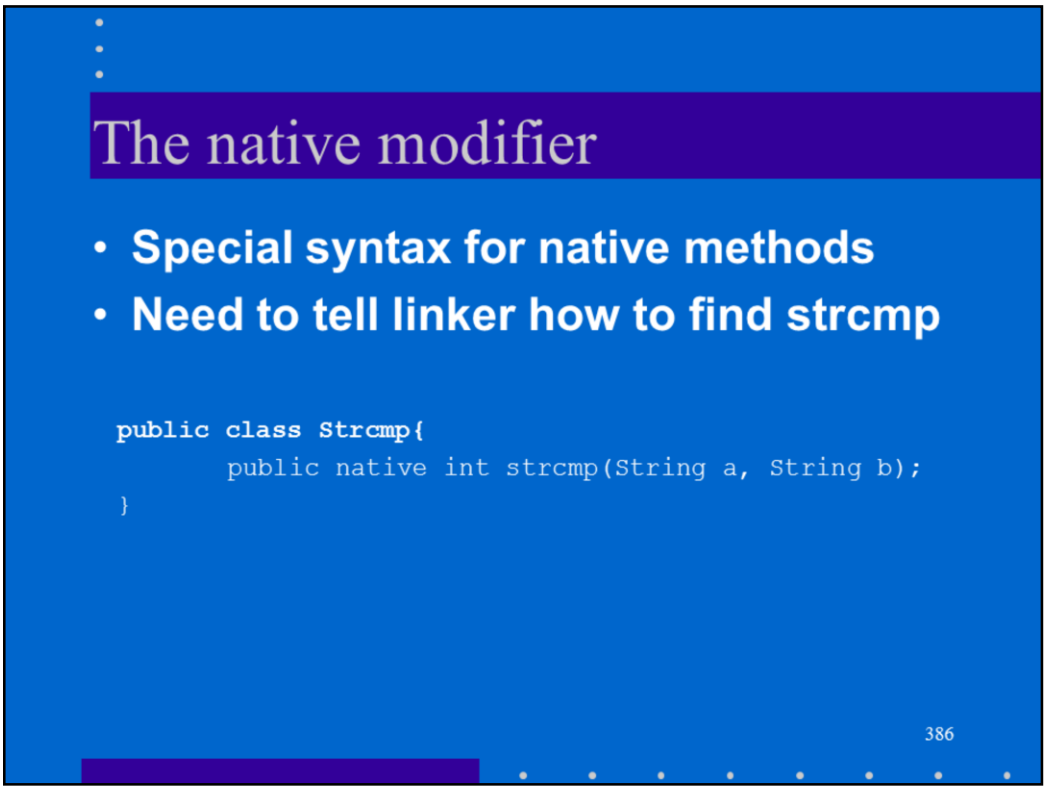
This said, it is very unlikely that we can write everything we need if we only program in Java.

It's clearly impossible to expect organisations to rewrite all their code in Java. Twenty years investment in software can't be discarded easily. Legacy software exists and often does it job very well. New applications in Java should be able to make use of it.

As yet, Java can't do everything. How do you make use of a joystick in a Java program? There are many tasks that can only be achieved by calling routines written in a language such as C or C++.

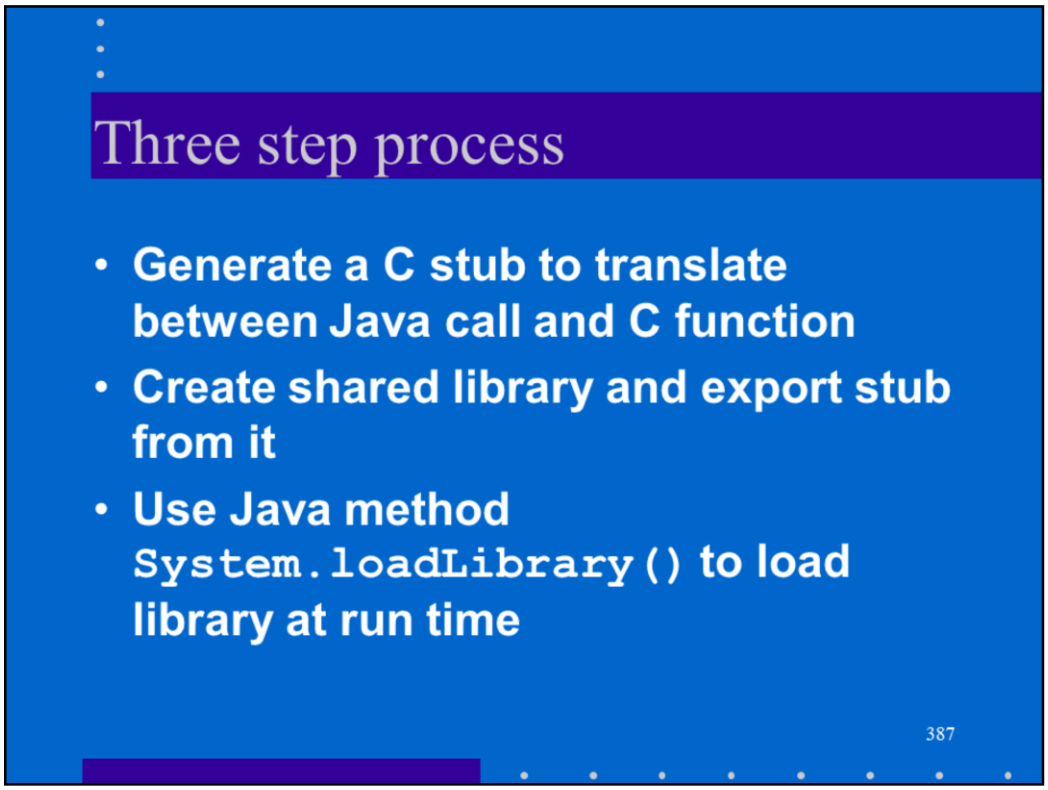
Although the comparison between the speed of execution of a Java program compared to a C program continues to become less significant as Java develops, there will still be the odd application where the speed of compiled code is essential.

It is essential, therefore, that Java provides mechanisms for calling out to routines written in other languages. Java calls such routines *native methods*. The use of native methods should be limited as they inhibit portability in Java programs.


The native modifier
• Special syntax for native methods
• Need to tell linker how to find strcmp
public class Strcmp{
public native int strcmp(String a, String b);
}
386

Java has a special syntax for native methods. The modifier `native` indicates that a method is implemented in the environment in which the program executes and is not written in Java.

Simply declaring that a method is a native method is not sufficient. The process whereby Java methods communicate with a native method is bound to be complicated. Java methods are translated into byte codes and these byte codes are interpreted at run time. Native methods will have been compiled into the machine code of the processor they are designed to run on. Java operates on objects whereas native method manipulate bits and bytes. Java needs to know where the code for the native methods is stored, how it should pass parameters to the method, and how it should interpret the results from the method.

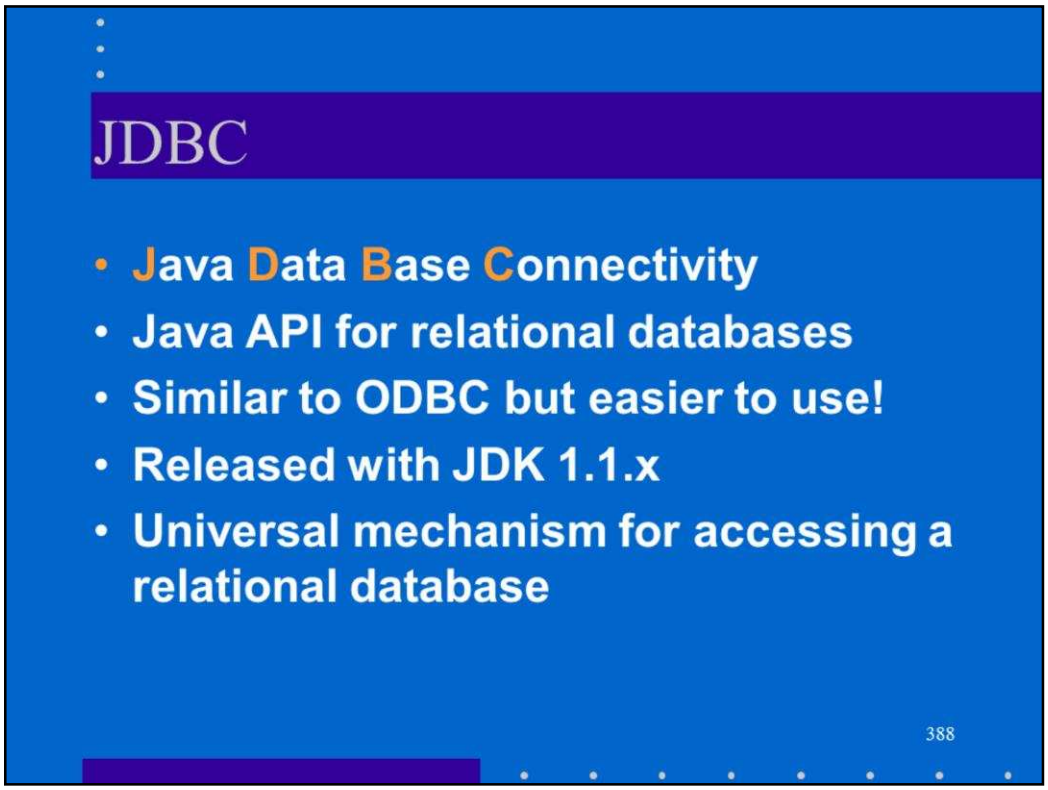

Three step process
• Generate a C stub to translate between Java call and C function
• Create shared library and export stub from it
• Use Java method System.loadLibrary() to load library at run time
387

The slide above describes the steps one might take to communicate between a Java program and a C function.

The first stage is to write a C routine which follows all the Java native code conventions. Within this there will be a call to the library routine that you want to make use of. The JDK contains a pre-processor which will generate a C/C++ include file for the code.

The code must then be compiled as a dynamically loaded library. This is where the include file comes in useful as it contains declarations which ensure that the library has the correct interface.

The Java program which uses the C function must make sure that the appropriate dynamically loaded library has actually been loaded before the function is called. It does this by using the method `System.loadLibrary()`.

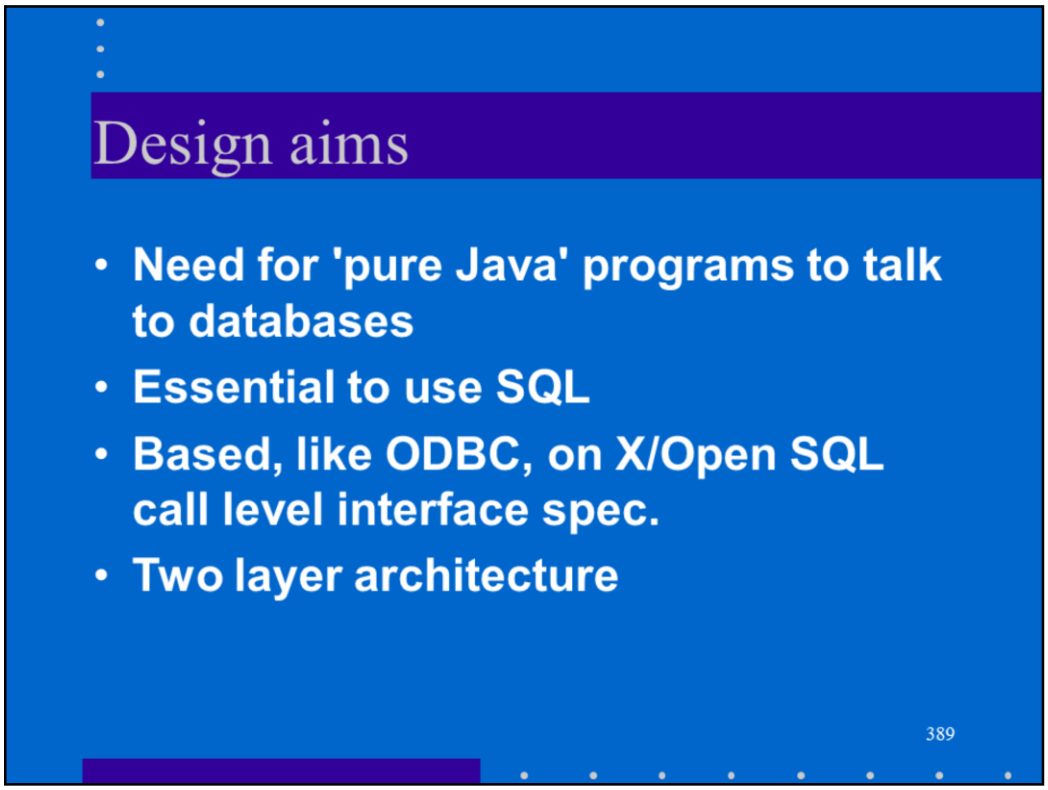

JDBC
• Java Data Base Connectivity
• Java API for relational databases
• Similar to ODBC but easier to use!
• Released with JDK 1.1.x
• Universal mechanism for accessing a relational database
388

In commercial data processing one of the main features of a language is its ability, or otherwise to store and retrieve data from a database. Currently, the major issue is whether the language can run SQL against a relational database.

Shortly after JDK 1.0 was released Sun announced the JDBC API which is a mechanism whereby Java programs can manipulate relational data from an extensive variety of relational database management systems. This was then incorporated into the 1.1.x release of the JDK as a standard package.

Microsoft had already released a technology which allowed programs written in a variety of languages to access data from a number of different data sources. This technology is known as ODBC (Open Data Base Connectivity). The underlying ODBC API is designed for C programmers. For the same reasons that programming in Java is easier than programming C, using JDBC is easier than using ODBC.

The idea of JDBC is "data anywhere". Regardless of the machine the program is running on, or the make of relational database management system the program is using, the code remains the same.

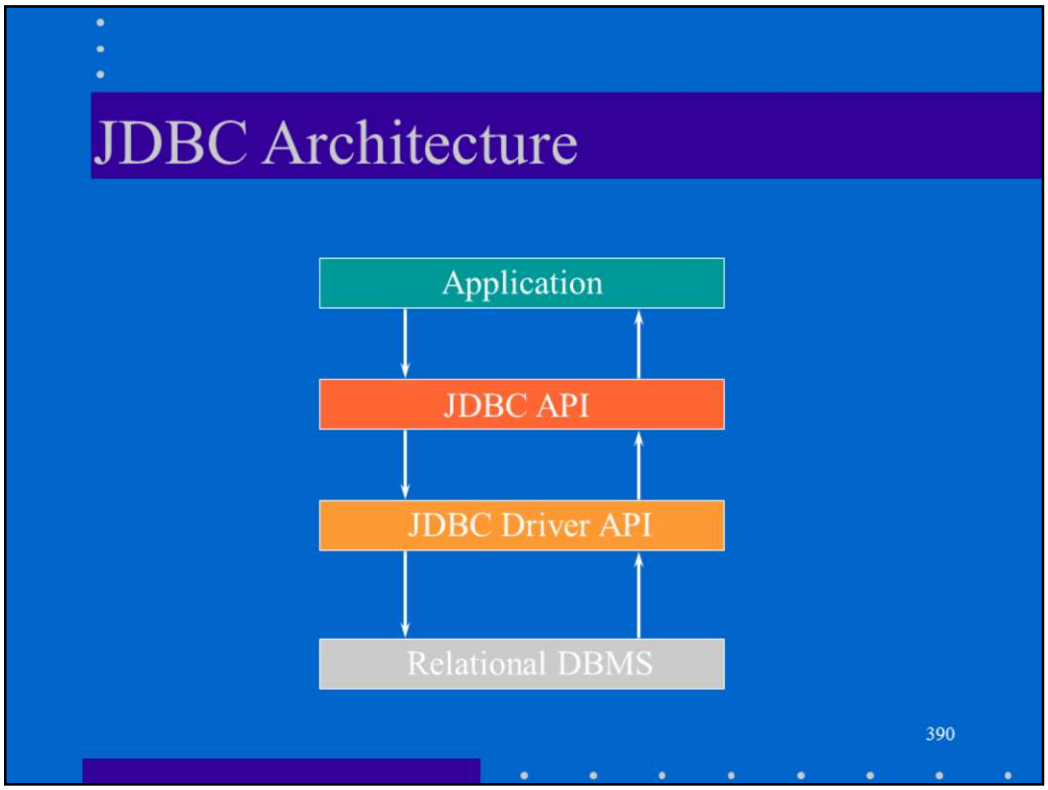

Design aims
• Need for 'pure Java' programs to talk to databases
• Essential to use SQL
• Based, like ODBC, on X/Open SQL call level interface spec.
• Two layer architecture
389

Use of JBDC means that pure Java programs don't have to call native methods to access relational data. This means that it is possible for an organisation which has a large variety of hardware and a centralised relational database management system to move very easily to a client-server implementation. The same program can run on all the various client platforms against data on a central server.

SQL has been described by a relational database industry guru (Michael Stonebraker) as 'intergalactic dataspeak'. Virtually all commercial relational databases support an SQL interface. JDBC and ODBC both support an underlying SQL implementation.

The idea of ODBC came from a portability standards group known as X/Open. They developed the specification of a DBMS independent mechanism for accessing relational data. JDBC, like ODBC, uses this specification.

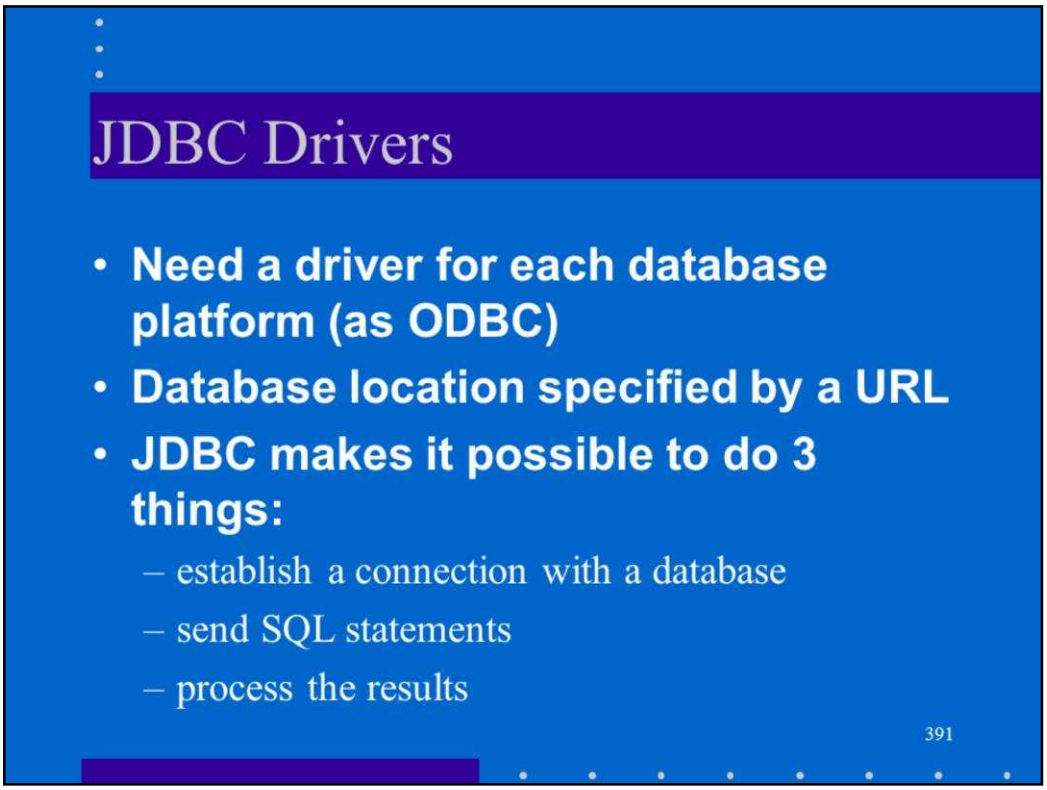
Clearly not all DBMS products are the same. Nor do they all implement the same dialect of SQL. JDBC uses a two level architecture to hide differences in the underlying DBMS platforms from the programmer.



Using JDBC an application (or applet) makes use of classes in the JDBC API. The JDBC API communicates with a JDBC Driver. Whereas the JDBC API is database independent (and identical across all hardware/software platforms) the driver is not. The driver communicates directly with a relational database product.

Programmers invoke standard JDBC methods. These are passed to the DBMS specific JDBC driver. The driver translates JDBC SQL into the version of SQL the DBMS understands. This SQL is then executed and the results are returned. The driver transforms these results into a form that the Java program can process.

The boxes marked JDBC API and JDBC Driver API represent the two level architecture.

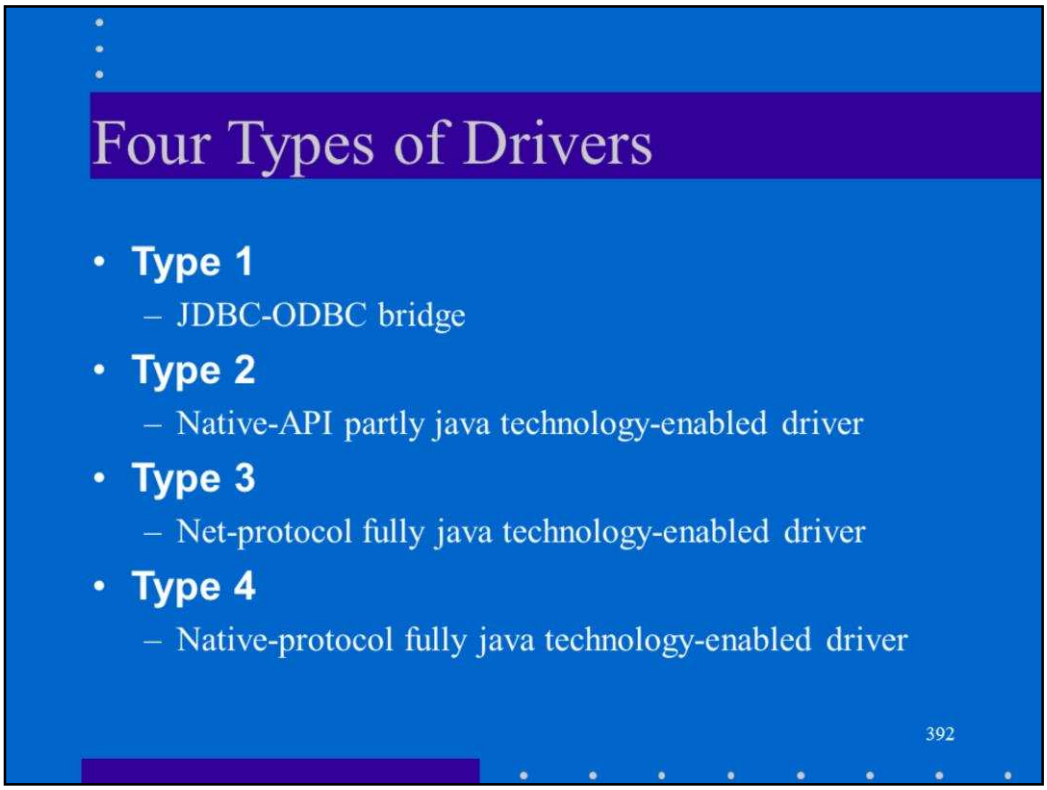

JDBC Drivers
• Need a driver for each database platform (as ODBC)
• Database location specified by a URL
• JDBC makes it possible to do 3 things:
– establish a connection with a database
– send SQL statements
– process the results
391

Essentially JDBC is ODBC for Java. It enables a Java programmer to query a relational database in a database independent manner. If a Java program works with a database implemented in Access, the same program will work unaltered with the same database design implemented in Oracle.

JDBC performs the same purpose as ODBC but it has been developed to be easier to use as it does not make heavy use of pointers.

A different driver is required for each different vendor's database. Normally these are supplied without cost by the database vendor. If an ODBC driver is available this can be used via the JDBC/ODBC bridge which enables a Java program to access a database that is ODBC compliant. A custom driver is preferable to this option as it reduces the number of software layers between the program and the data.

At connection time, a program using JDBC may specify a remote database. The mechanism for specifying the location of the database resembles a standard URL.



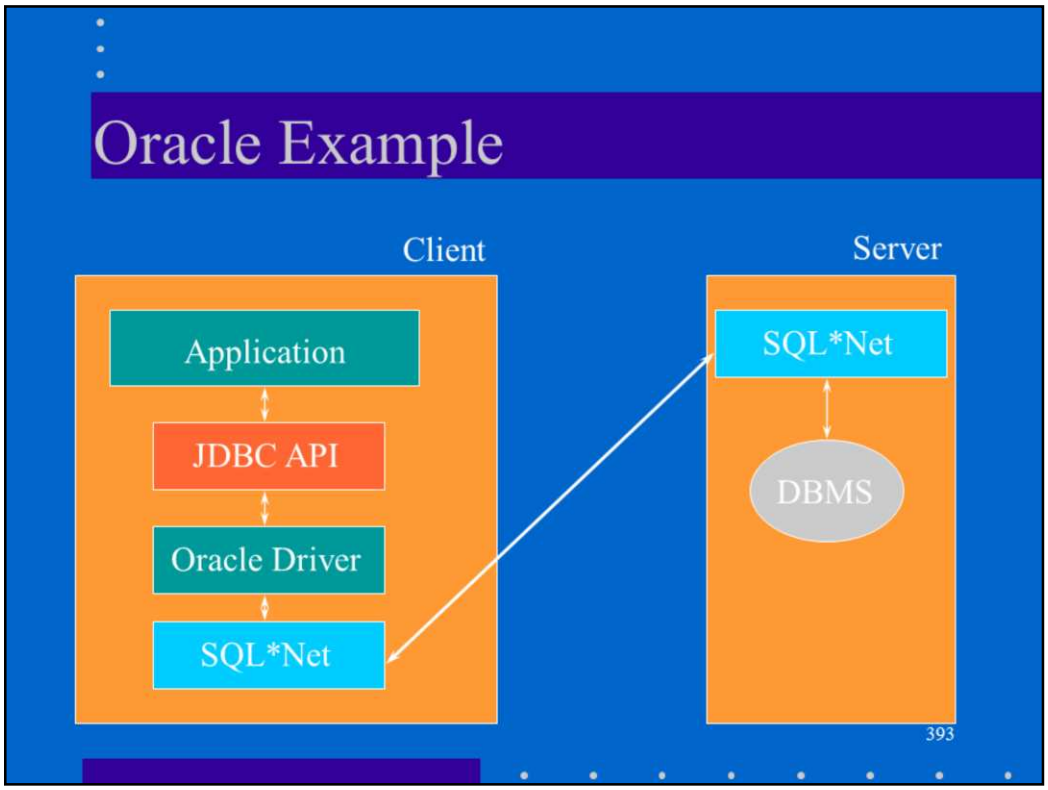
From http://java.sun.com/products/jdbc/driverdesc.html

JDBC technology drivers fit into one of four categories:

1. A *JDBC-ODBC bridge* provides JDBC API access via one or more ODBC drivers. Note that some ODBC native code and in many cases native database client code must be loaded on each client machine that uses this type of driver. Hence, this kind of driver is generally most appropriate when automatic installation and downloading of a Java technology application is not important.
2. A *native-API partly Java technology-enabled driver* converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.
3. A *net-protocol fully Java technology-enabled driver* translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect all of its Java technology-based clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC API alternative. It is likely that all vendors of this solution will provide products suitable for Intranet use. In order for these products to also support Internet access they must handle the additional requirements for security, access through firewalls, etc., that the Web imposes. Several vendors are adding JDBC technology-based drivers to

their existing database middleware products.

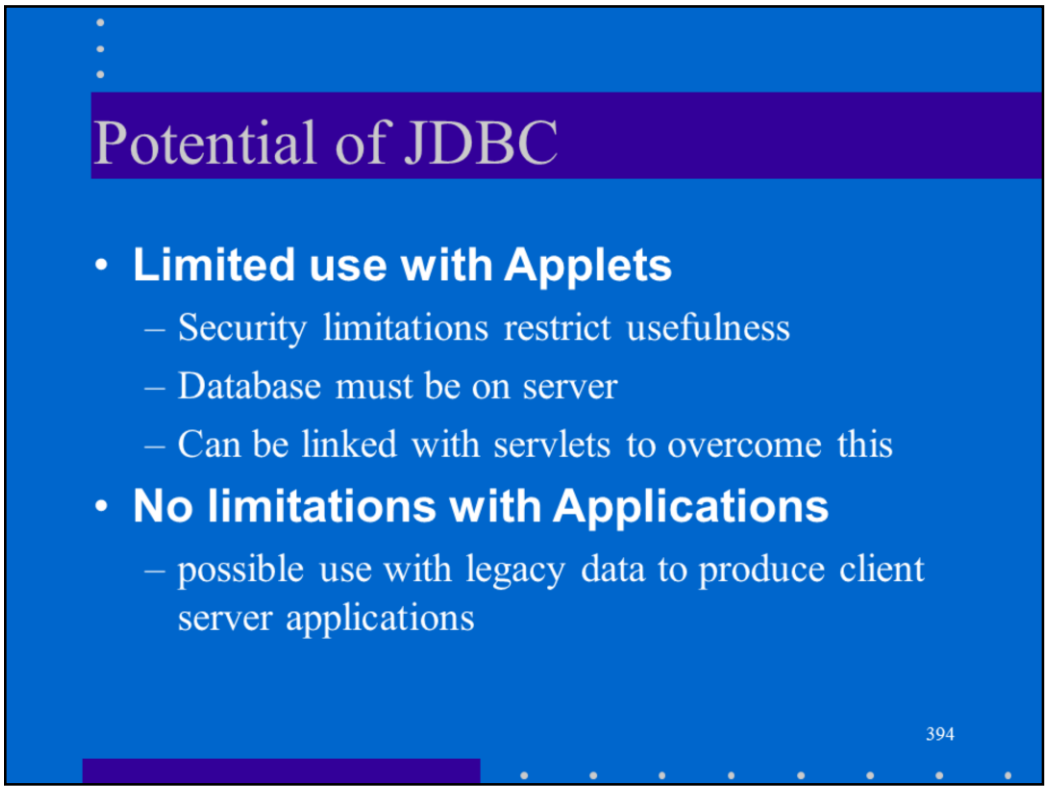
4. A *native-protocol fully Java technology-enabled driver* converts JDBC technology calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver. Several database vendors have these in progress.



This diagram shows the use of a type 2 driver. The database management system shown here is Oracle which is one of the market leaders. A Java application contains calls to JDBC methods. Typically these methods will accept strings containing SQL in the JDBC dialect.

The SQL is passed on to the Oracle JDBC driver. The driver is distributed free on the Internet and translates JDBC SQL into Oracle's dialect of SQL. This driver then communicates with Oracle's standard network handling product called SQL*Net. The query is transmitted across the network and executed on the server.

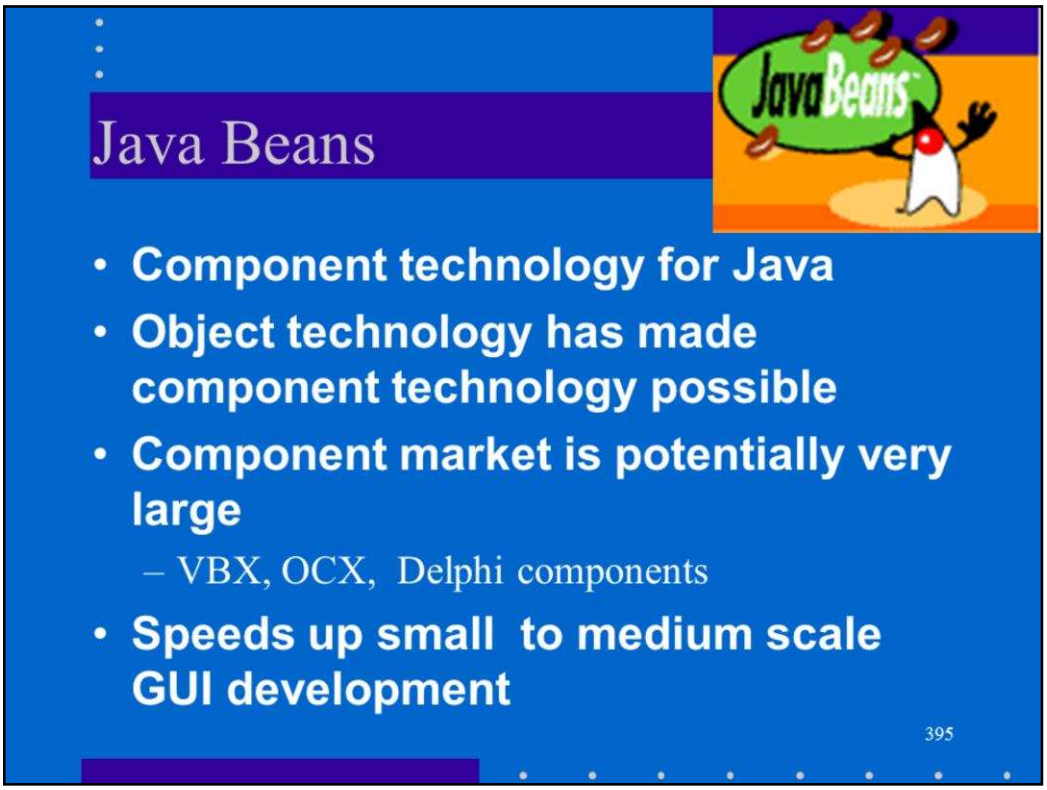
The results are transmitted across the network and made available to the Java program.


Potential of JDBC
• Limited use with Applets
– Security limitations restrict usefulness
– Database must be on server
– Can be linked with servlets to overcome this
• No limitations with Applications
– possible use with legacy data to produce client server applications
394

Ostensibly JDBC could be very useful. It should be possible to download database applets to any Internet connected machine and use a database remotely. But there are drawbacks. To use JDBC in an applet either JDBC must be built into the browser or else it must be downloaded with the applet. The first of these solutions could lead to browser bloat (and this has now become an issue), the second would lead to long download times. In addition because of the security rules which apply to applets the database could only reside on the Web server and nowhere else. This is not normal practice. The answer may be to use Java servlet technology which is covered later in this lecture.

Applications, on the other hand, do not suffer from the same limitations and may access data from relational databases located anywhere on the network. Keeping releases of applications up to date is, however, more difficult than with applets which are downloaded on demand. In addition, in order for an applet to run a JVM must be installed on the client computer.

Details of JDBC are available at:
http://java.sun.com/products/jdbc/index.html


Java Beans
JavaBeans
• Component technology for Java
• Object technology has made component technology possible
• Component market is potentially very large
– VBX, OCX, Delphi components
• Speeds up small to medium scale GUI development
395

From the JavaSoft's Frequently Asked Questions on Beans:

Why are component architectures useful?

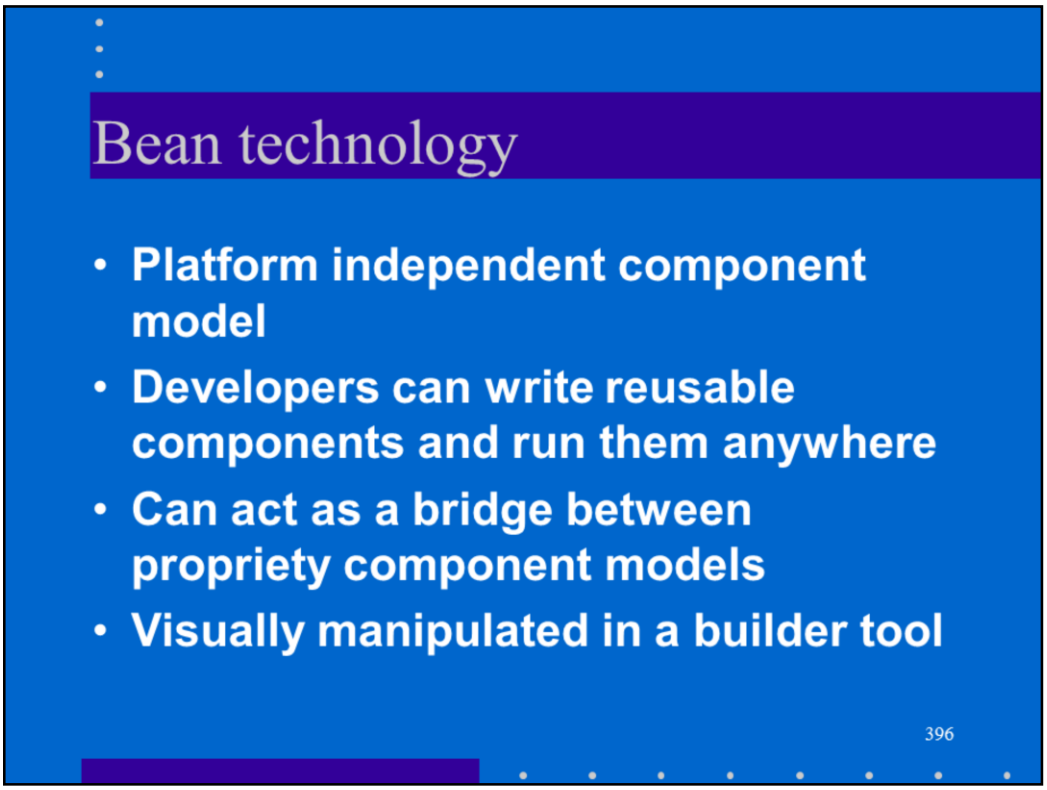
Developers are turning to creating components rather than monolithic applications to free themselves from slow, expensive application development, and to build up a portable, reusable code base. This enables developers to quickly attack new market opportunities, new joint development opportunities, and new ways to sell smaller packages of software.

Is JavaBeans a complete component architecture?

JavaBeans is a complete component model. It supports the standard component architecture features of properties, events, methods, and persistence. In addition, JavaBeans provides support for introspection (to allow automatic analysis of a JavaBeans component) and customisation (to make it easy to configure a JavaBeans component).

Why a component architecture for the Java platform?

JavaBeans brings the extraordinary power of the Java platform to component development, offering the ideal environment for a developer who wants to extend the concept of reusable component development beyond one platform and one architecture to embrace every platform and every architecture in the industry.

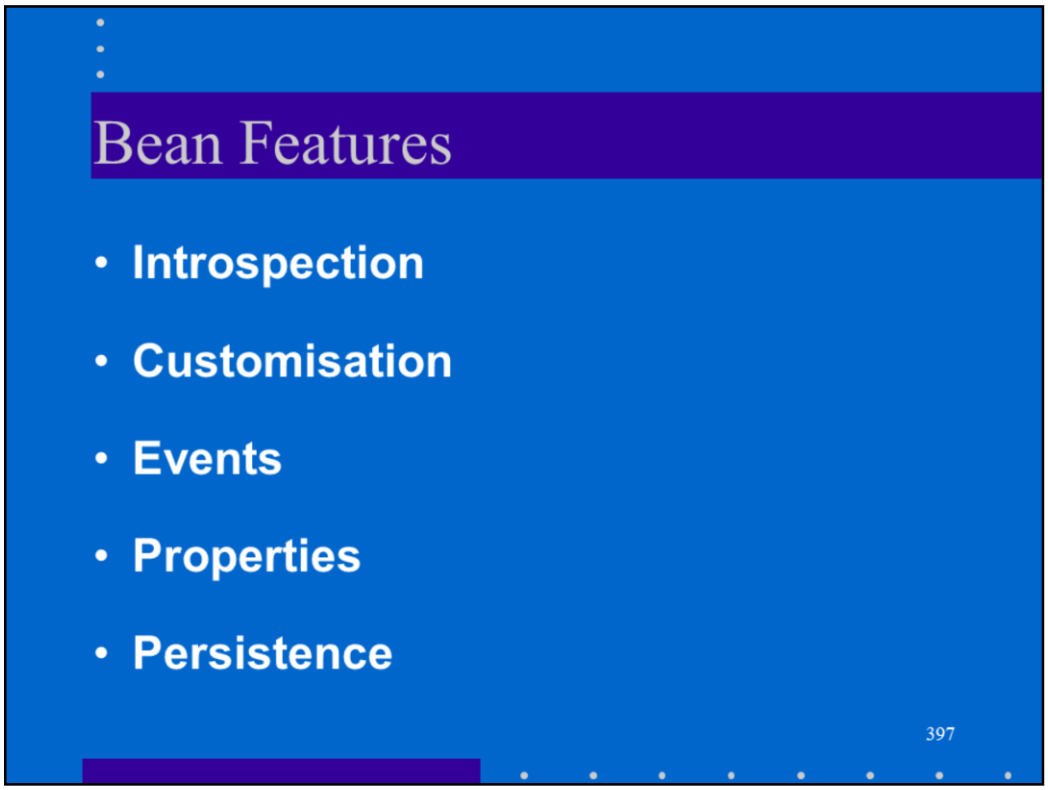

Bean technology
• Platform independent component model
• Developers can write reusable components and run them anywhere
• Can act as a bridge between propriety component models
• Visually manipulated in a builder tool
396

Microsoft has demonstrated the advantages of component technology via VBX, OCX and finally ActiveX. Microsoft components, however, can only be used on Intel platforms (or Alpha chips) which run Windows. Java beans offer a similar technology but one which is portable.

Beans are a way for a programmer to produce components which can be reused and can be used across different platforms. Examples of typical beans are buttons, image panels etc.

Beans can be used as wrappers around ActiveX components so as to allow Java programmers access to Microsoft controls.

The great advantage of Java Beans is that they make programming in Java similar to programming in Visual Basic. Visual elements are dragged into a builder and are then connected together to construct an applet or application.



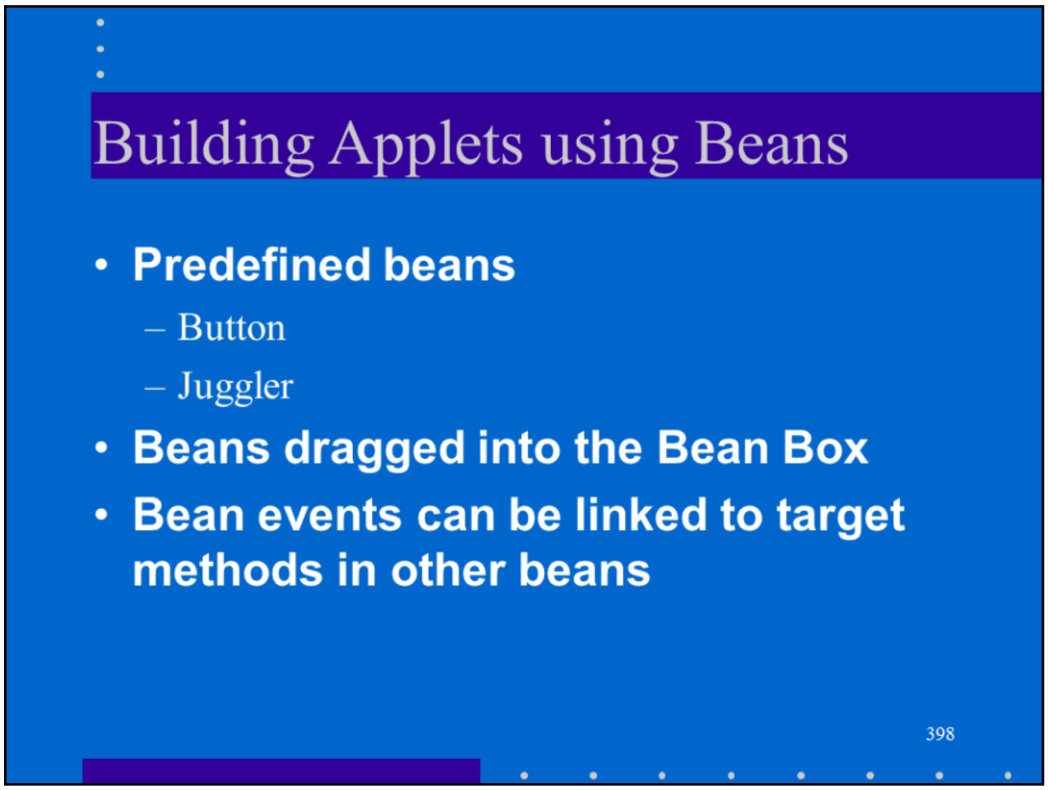
Introspection: enables a builder tool to analyse how a Bean works

Customisation: enables a developer to use an application builder tool to customise the appearance and behaviour of a Bean

Events: enables Beans to communicate and connect together

Properties: enable developers to customise and program with Beans

Persistence: enables developers to customise Beans in an app builder, and then retrieve those Beans, with customised features intact, for future use


Building Applets using Beans
• Predefined beans
– Button
– Juggler
• Beans dragged into the Bean Box
• Bean events can be linked to target methods in other beans
398

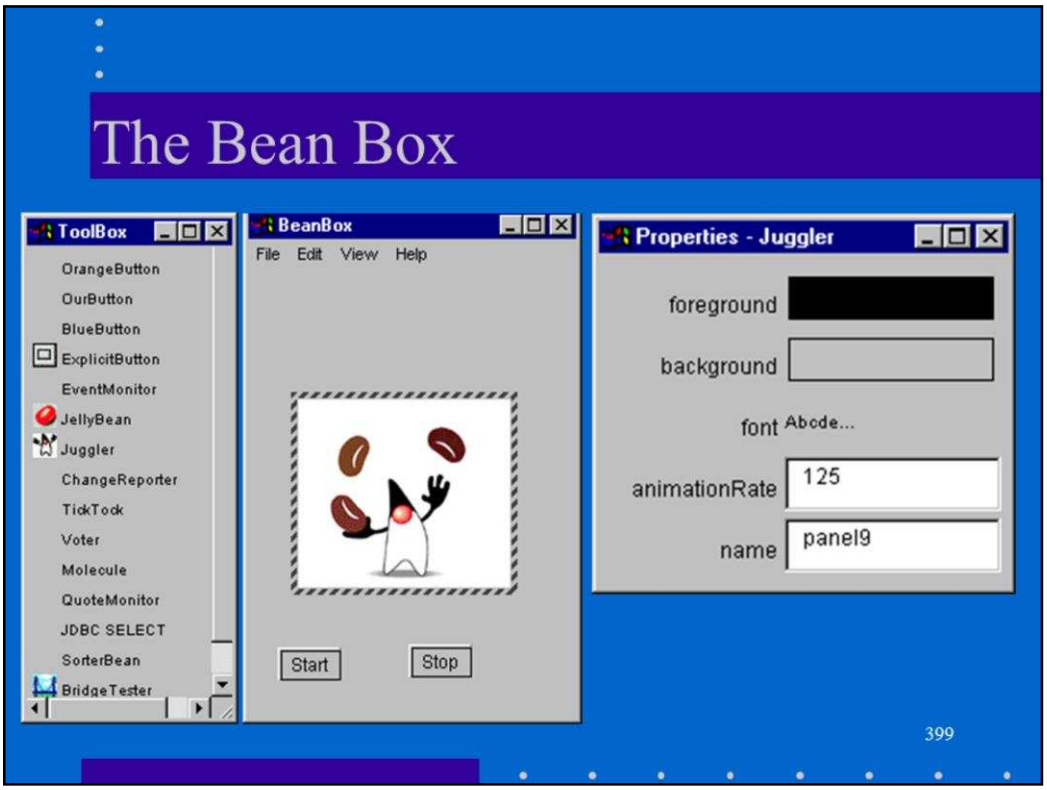
Essentially, beans are Java classes, written in a standard way, which makes it possible for them to be manipulated in a visual environment. Anyone who can write code for a class can write a bean.

The standard examples included in the Bean Development Kit (BDK) include a Button and a Juggler (an animation of the Java Duke juggling beans).

With the BDK Sun provide a primitive development environment called the Bean Box. This is regarded as a ‘proof of concept’ release, Sun expect that other companies will release more robust programming environments.

Beans are dragged from the Tool Box into the Bean Box. Each bean has some properties which may be set in the Properties box.

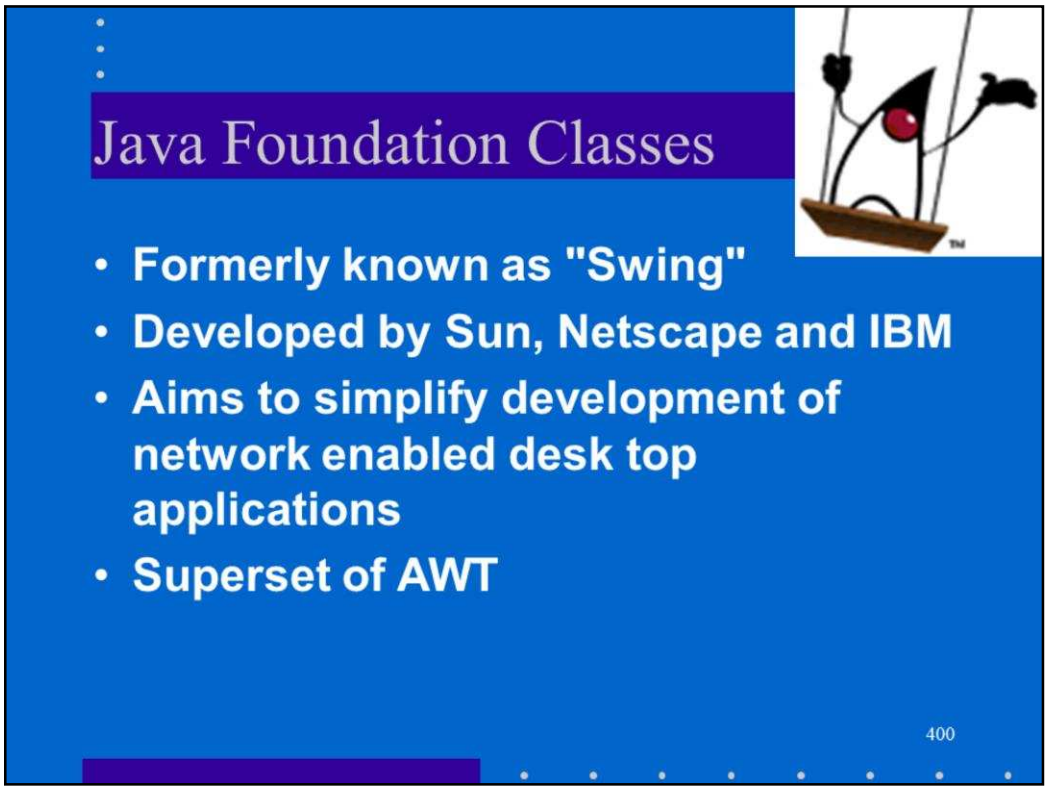
Beans make events and methods visible to the programmer constructing the applet (the programmer can right click on a bean to see these). An event in one bean may be visually connected to a method on another bean. The effect is to construct code where if, during execution, the selected event occurs then the method linked to that event will be invoked.



This is a picture of an applet being constructed using the Bean Box. Three beans are in the Bean Box, two buttons and a juggler. The juggler is currently selected and the right hand window shows the properties that may be set for it. The applet has been constructed so that pressing the start button causes the juggler to start juggling and the stop button causes him to stop. This is achieved by connecting the ActionPerformed event of the button to the start and stop methods implemented by the juggler bean. No code is necessary.

Details of JavaBeans are available at: http://java.sun.com/beans/index.html

The JBK can be downloaded (free!) from:
http://java.sun.com/beans/software/index.html


Java Foundation Classes
• Formerly known as "Swing"
• Developed by Sun, Netscape and IBM
• Aims to simplify development of network enabled desk top applications
• Superset of AWT
400

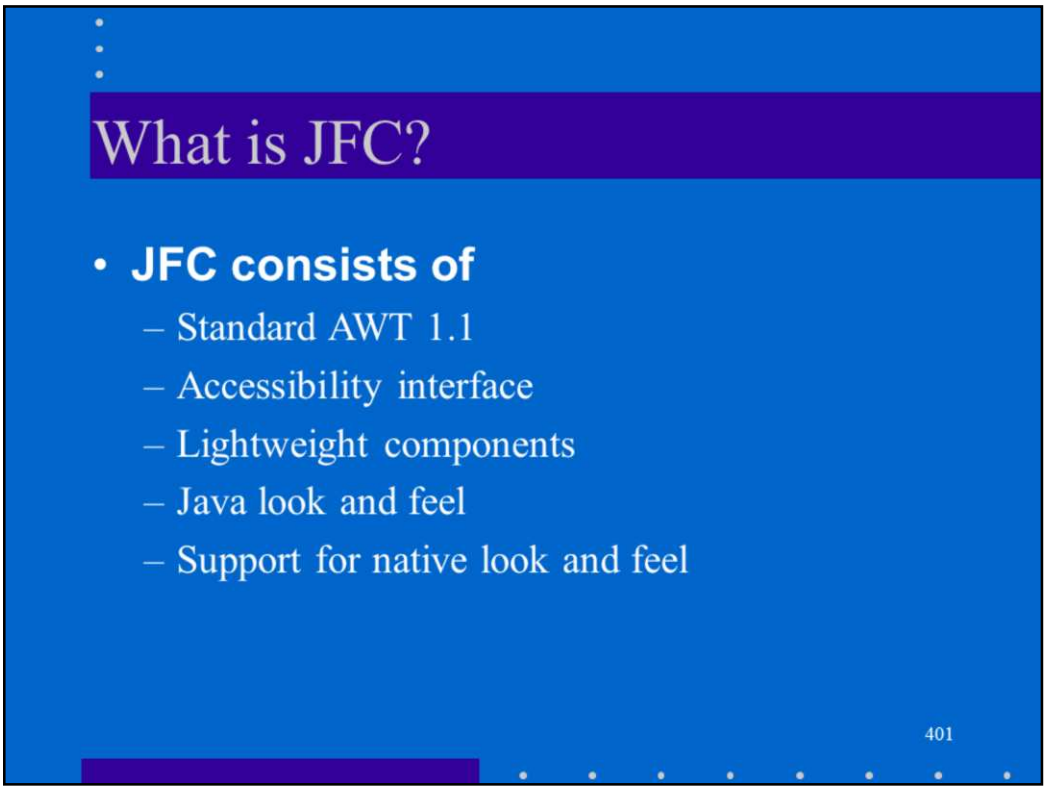
"The Java Foundation Classes (JFC) are a set of Java class libraries provided as part of J2SE™ to support building graphics user interface (GUI) and graphics functionality for Java technology-based client applications ("Java applications"). JFC includes an extensive set of technologies that enable developers to create a ***rich, interactive user experience*** for client applications that can run not only on Microsoft Windows, but also on other increasingly popular platforms, such as Mac OSX and Linux."

One of the most criticised aspect of Java at the time of the release of JDK1.1 was the AWT. The AWT, as it stands, is not really sufficient for building GUI applications that compare with those available under most windowing platforms.

As a response to these criticisms, Sun, IBM and Netscape developed a set of classes under the code name "Swing". These classes are a superset of those available under AWT and are reckoned by most Java programmers to be vastly superior to standard AWT. Theses classes have now been renamed the Java Foundation Classes (JFC).

Details of JFC are available at:http://java.sun.com/products/jfc/index.html

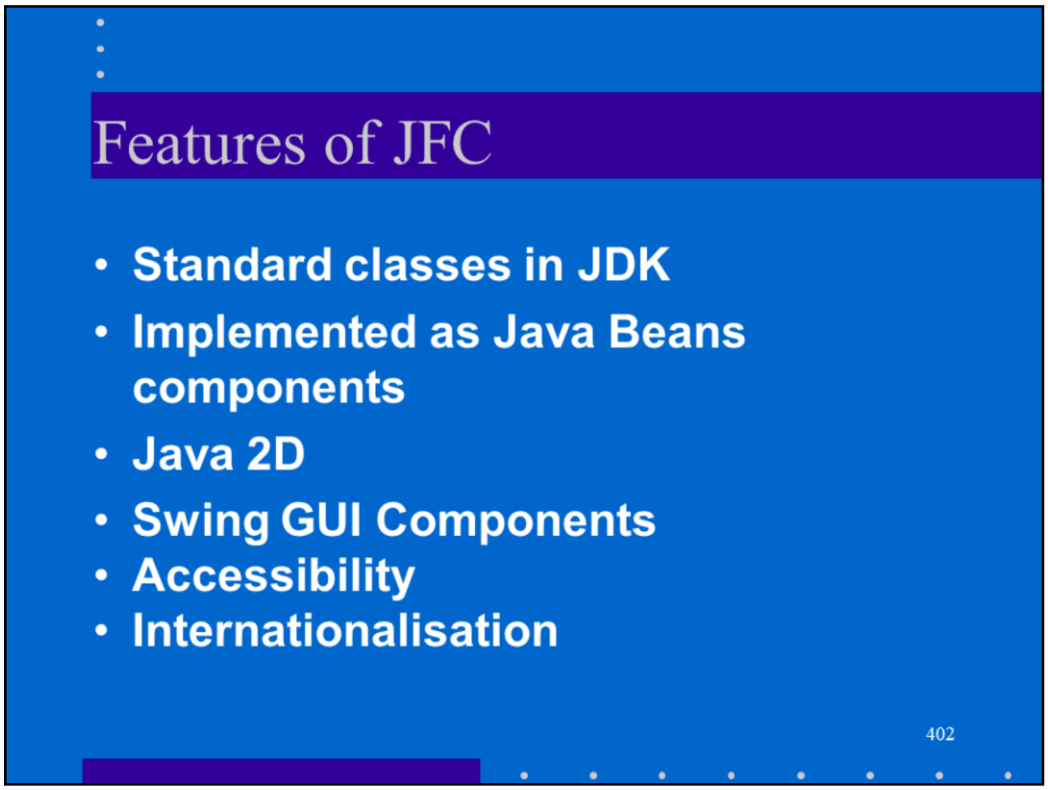
These classes are now standard in Java 2.



As part of JFC, the Accessibility API enables Java applications to work with alternative input and output devices such as Screen Readers, Screen Magnifiers, Braille terminals, and others. The Accessibility API takes the current Java system that is visually oriented and extends it to be usable by people who need to work with other non-visual devices.

Lightweight components are user interface components that do not subclass an existing AWT interface element. They do not use native interface elements as provided by the underlying windowing system. This means that they are less limiting than standard AWT components.

Sun says: "Applications built using JFC will not be locked in to a specific look and feel. Using JFC, developers can create apps that either have a native platform look and feel or use the new Java look and feel -- or they can create their own custom look and feel. The power lies in the hands of developers -- they can develop their products using the look and feel that is most appropriate for their users."



Abstract Window Toolkit (AWT): APIs that enable programs to integrate into the native desktop window system, including APIs for Drag and Drop.

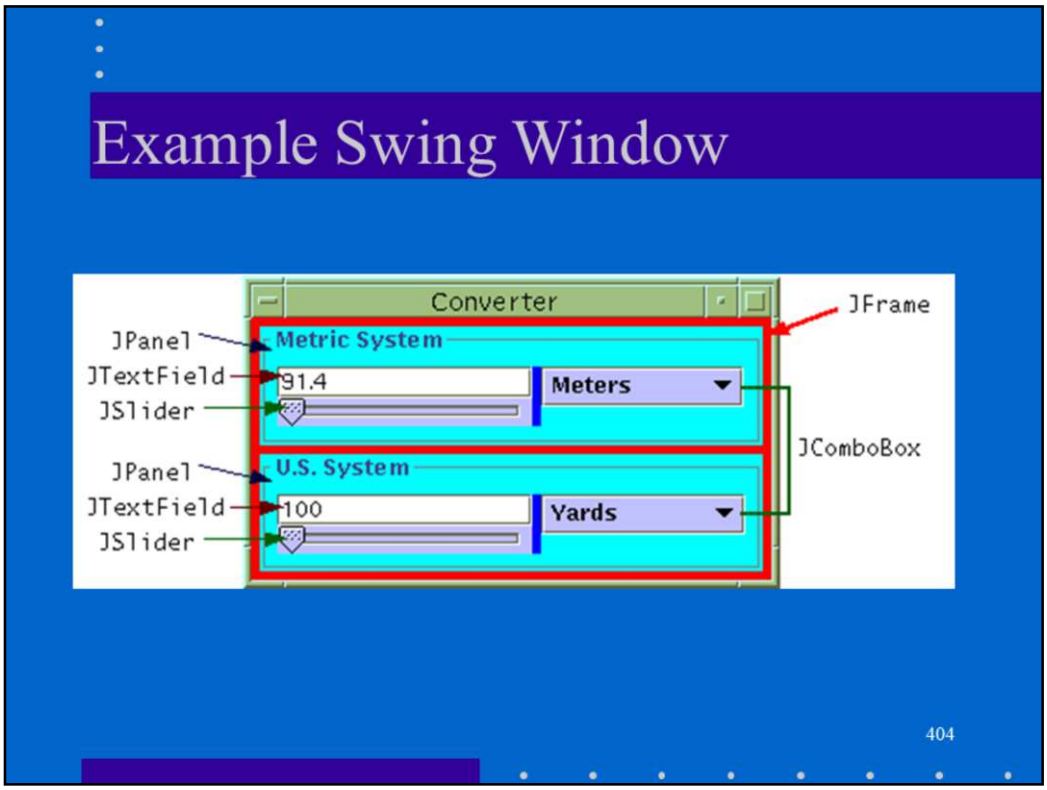
Java 2D™: APIs to enable advanced 2D graphics, imaging, text and printing.

Swing GUI Components: APIs that extend the AWT to provide a rich, extensible GUI component library with a Pluggable look and feel.

Accessibility: APIs and assistive technologies for ensuring an application is accessible to users with disabilities and meets government requirements for accessibility.

Internationalization: All JFC technologies include support for creating applications that can interact with users around the world using the user's own language and cultural conventions. This includes the Input Method Framework API.

Documentation for these components is available at:
http://java.sun.com/products/jfc/swingdoc-api/overview-summary.html

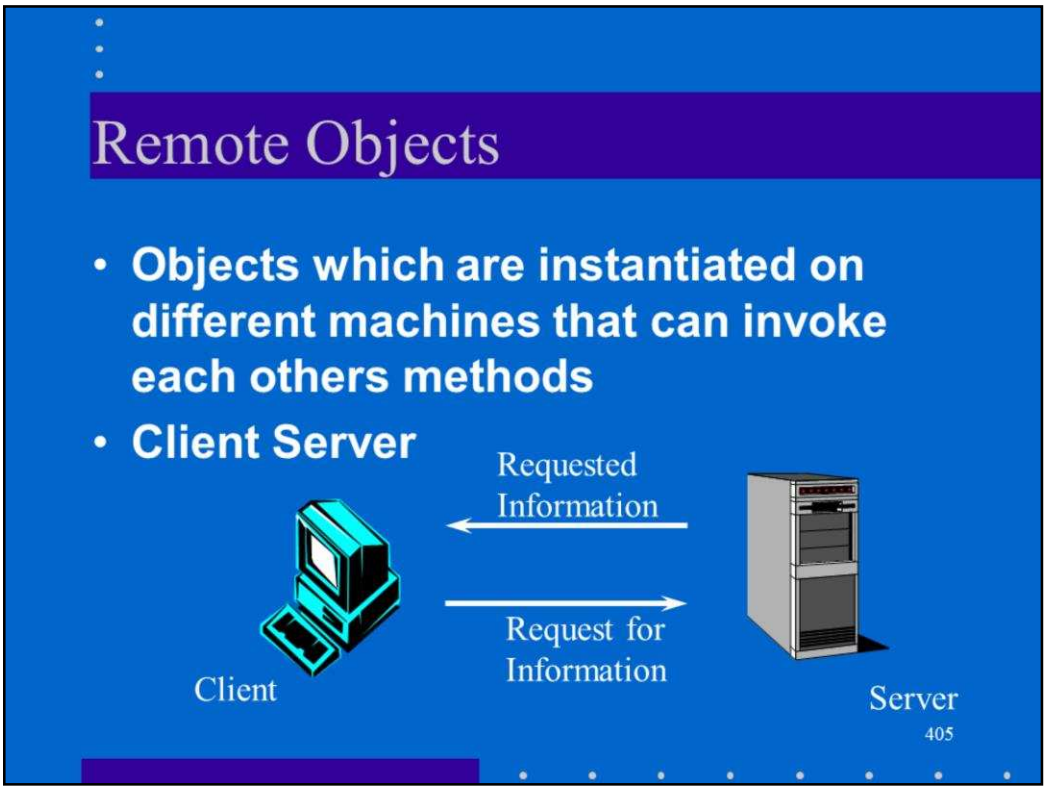


The graphic shows the output of a program which uses Swing components. The look and feel is the Java default look and feel.

“The Swing toolkit is a fully-featured UI component library implemented entirely in the Java programming language. J2SE's javax.swing package uses the windowing functionality of AWT and the rendering capabilities of Java 2D to provide sophisticated and highly extensible UI components that comply with the JavaBeans™ specification.

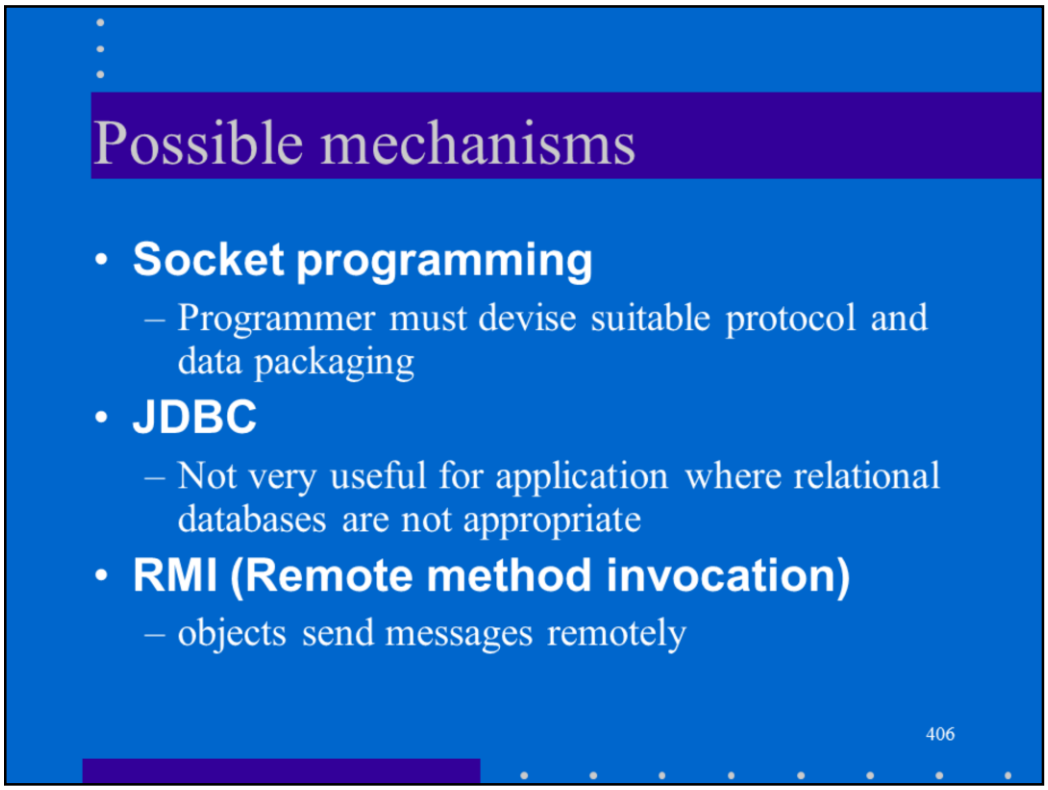
The Swing components use other JFC features, such as drag and drop, the accessibility APIs, and J2SE internationalization support to create fully featured UI functionality out of the box.

Swing also includes the javax.swing.text package, which contains support for implementing sophisticated styled-text rendering and editing, as well as support for rendering and mutating HTML documents. “



When one object invokes another object's methods it is normal for both objects to be on the same machine. It is possible, however, in Java to invoke a method of an object which is located on another machine. The invocation will look identical to the invocation of a local object method, but the JVM will redirect the call over the network and return the results when they are available. This makes it possible to implement distributed objects.

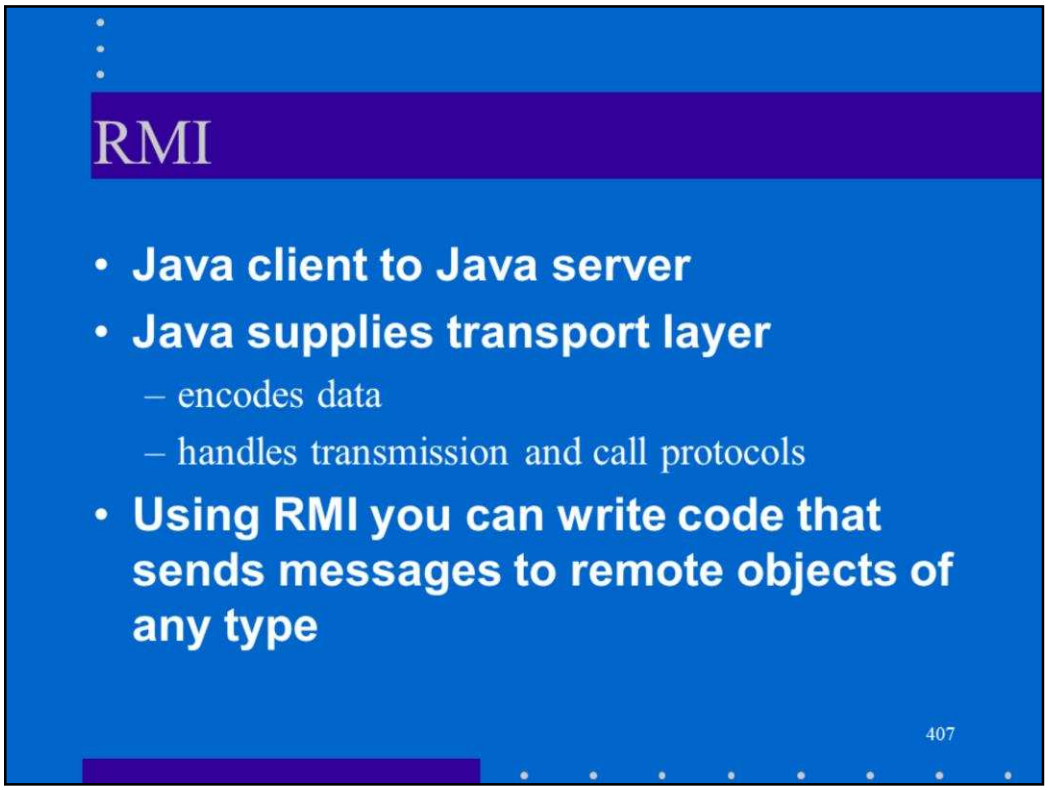
Mechanisms for distributing objects also exist in the CORBA standard (http://www.omg.org), the Java solution is more lightweight and whilst CORBA allows objects written in a variety of languages to communicate the Java mechanism only works for objects coded in Java.


Possible mechanisms
• Socket programming
– Programmer must devise suitable protocol and data packaging
• JDBC
– Not very useful for application where relational databases are not appropriate
• RMI (Remote method invocation)
– objects send messages remotely
406

There are a number of ways in which a Java program could talk to a remote object. The normal communication mechanism in Java is via TCP/IP sockets. This mechanism is, however, far more complicated than a normal method invocation. To use sockets to remotely invoke a method would require significant amounts of extra coding at both the caller and the called ends of the communication.

JDBC is, in effect, a type of remote method invocation when the database is remote from the application. The methods that may be invoked by JDBC are however limited to those that deal with relational databases.

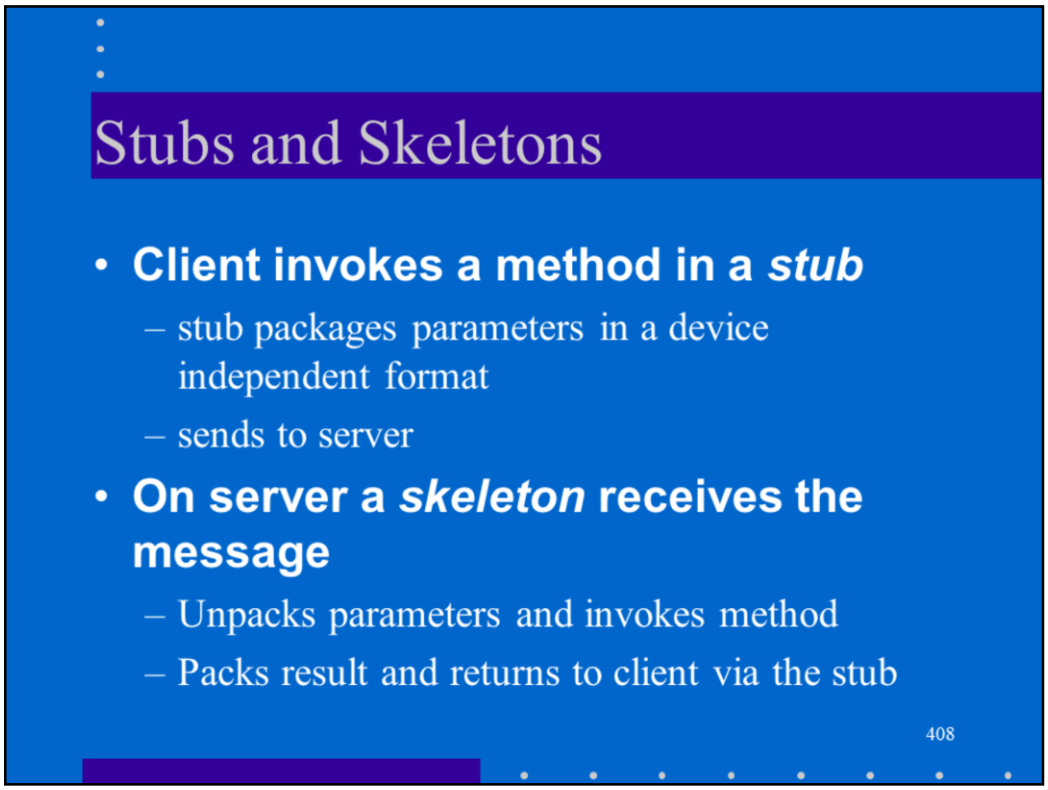
Java provides a third mechanism, RMI (remote method invocation), which enables a programmer to invoke a method for an object which is on a remote machine, in a manner identical to a local method invocation.


RMI
• Java client to Java server
• Java supplies transport layer
– encodes data
– handles transmission and call protocols
• Using RMI you can write code that sends messages to remote objects of any type
407

The terminology used is that a Java client invokes a method on a Java server. This does not imply, however, that it is only possible to invoke methods of objects stored on a server machine. A program which invokes a remote method is called a client regardless of its physical location. Similarly a server is a program which responds to a remote method invocation. It is possible that in their next interaction client and server will switch roles.

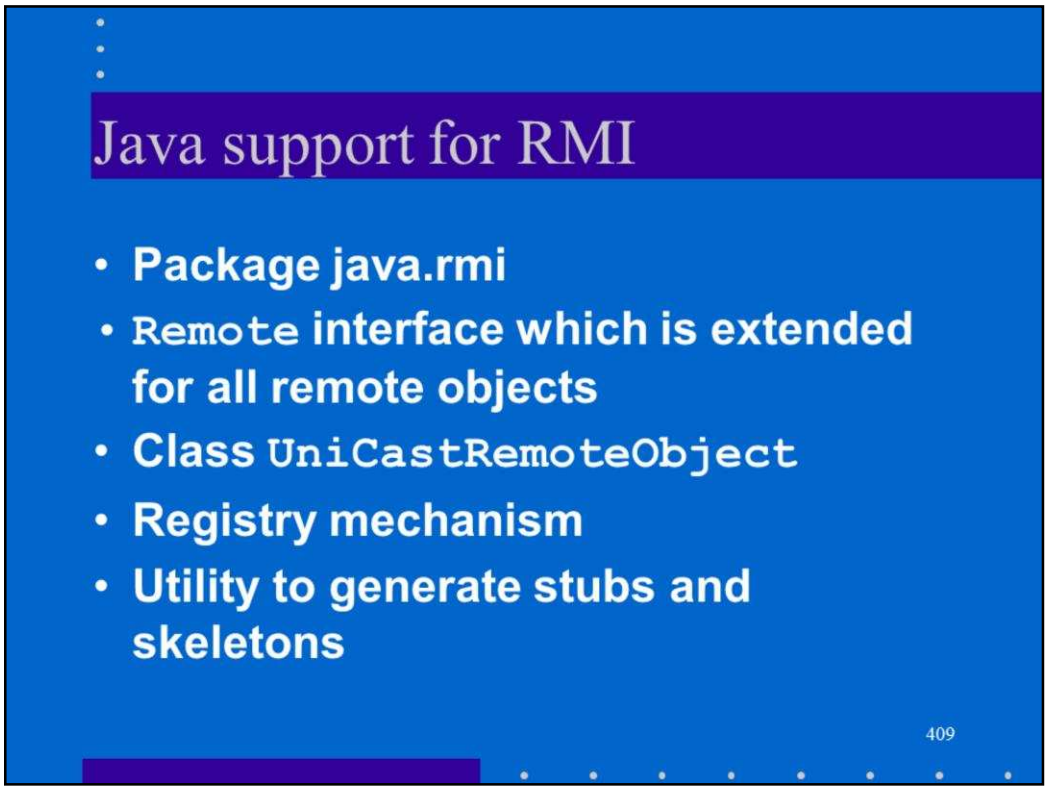
The Java system takes the responsibility of packaging up any parameters passed to the called method, transporting the parameters across a network link and then calling the remote method with the parameters. The result of the method is then packaged up and transmitted back to the caller.

Clearly this doesn't happen by magic, therefore the programmer is responsible for informing the system that a remote method invocation is required, as well as where the remote method resides. These additions take place, however, in code which is external to the code containing the method invocation.


Stubs and Skeletons
• Client invokes a method in a stub
– stub packages parameters in a device independent format
– sends to server
• On server a skeleton receives the message
– Unpacks parameters and invokes method
– Packs result and returns to client via the stub
408

The way RMI is implemented is that when a calling routine invokes a method rather than entering the code of the method it is calling, it enters code belonging to a stub method of the same name. The stub method deals with the communication with the remote site. It constructs a message which identifies the object to which the method is to be applied, adds in the parameters, and sends the message to the site where the object resides.

This message is received by a skeleton which unpacks the message and invokes the required method code on the specified objects. It receives its result from the invocation of the method, packages it up and passes it back to the stub on the calling machine. The stub unpacks the message and returns the result to the caller as though a local call had taken place.


Java support for RMI
• Package java.rmi
• Remote interface which is extended for all remote objects
• Class UniCastRemoteObject
• Registry mechanism
• Utility to generate stubs and skeletons
409

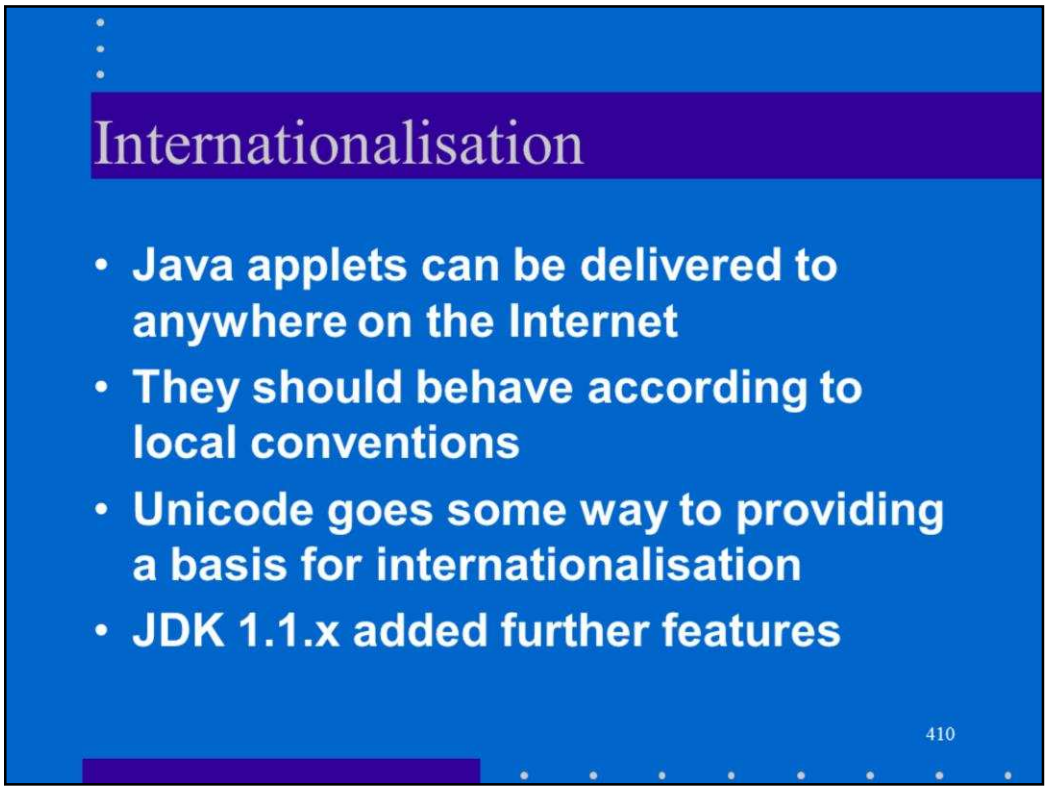
The package which contains the classes which make this possible is called `java.rmi`.

This package contains an interface `Remote` which must be extended by every class that can be remotely invoked. All the methods that may be remotely invoked will appear in this extended interface.

Server classes must inherit from the concrete class `UniCastRemoteObject`.

RMI requires a registry mechanism so that a client can obtain a reference to a remote object. This is implemented via methods of `UniCastRemoteObject`.

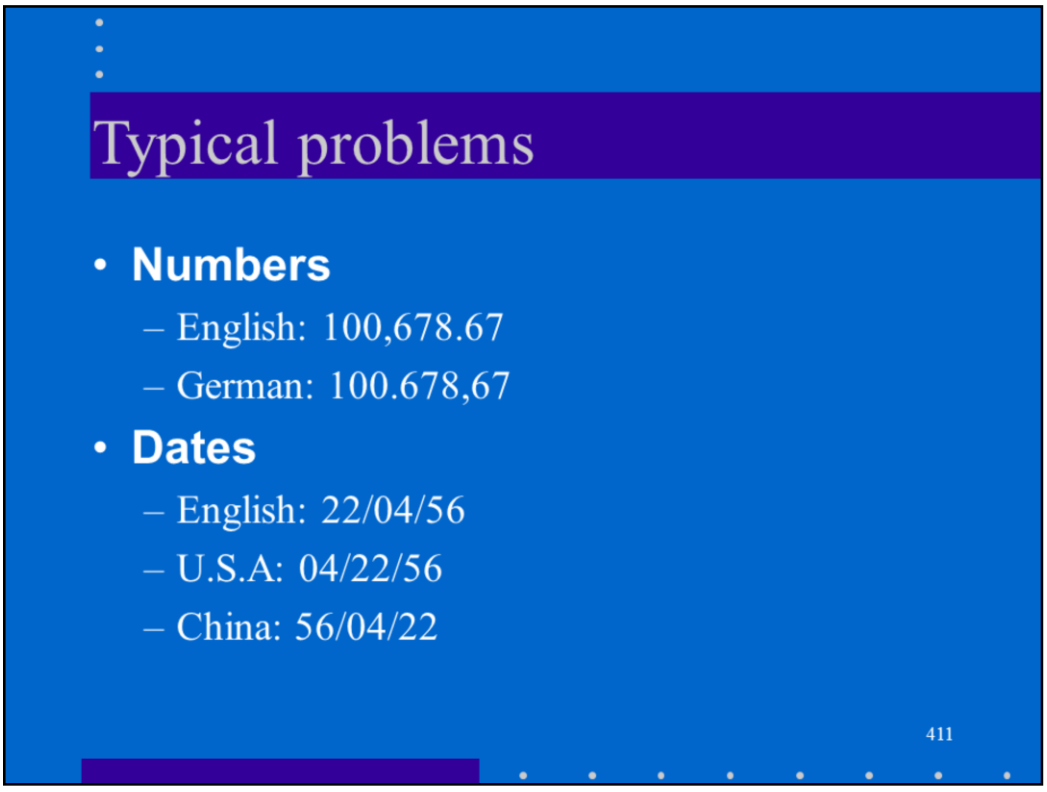
Stubs and skeletons are automatically generated automatically by the `rmic` tool which is supplied with the JDK.


Internationalisation
• Java applets can be delivered to anywhere on the Internet
• They should behave according to local conventions
• Unicode goes some way to providing a basis for internationalisation
• JDK 1.1.x added further features
410

Applets may be executed halfway across the world from the machine they reside on. If they are to be useful in such a context they need to adapt their output to the country in which they are executing.

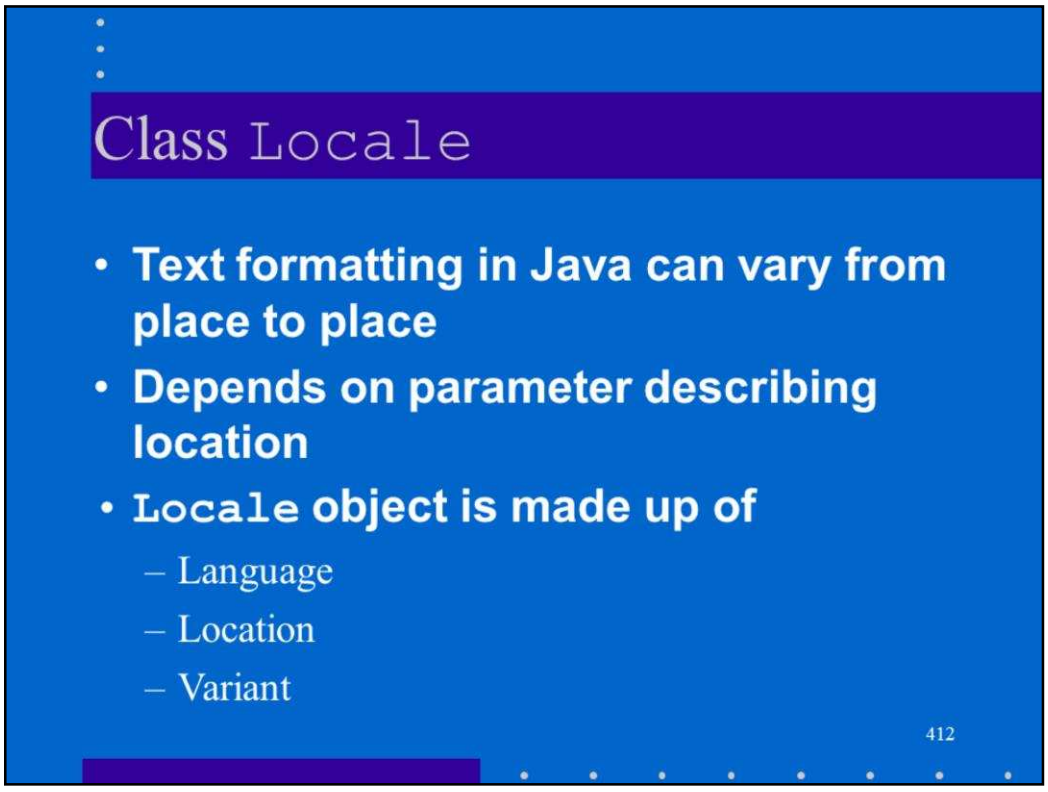
Aside from this, programming languages have for too long been firmly embedded in English/American culture and have largely ignored conventions outside of those accepted by Anglo-Americans. To truly represent the various cultures of computer users, programming languages must be extended to deal with far more variation than they have in the past. Java is the first language which has made an attempt to be truly international.

The introduction of Unicode means that Java can deal with extended character sets, but this on its own is not sufficient to make it an international language. With the release of JDK 1.1.x further internationalisation features were added to Java.



The difference between cultures are quite surprising. In English we separate whole number numbers with a comma for readability and indicate decimal parts with a full stop (a decimal point). The German convention reverses this.

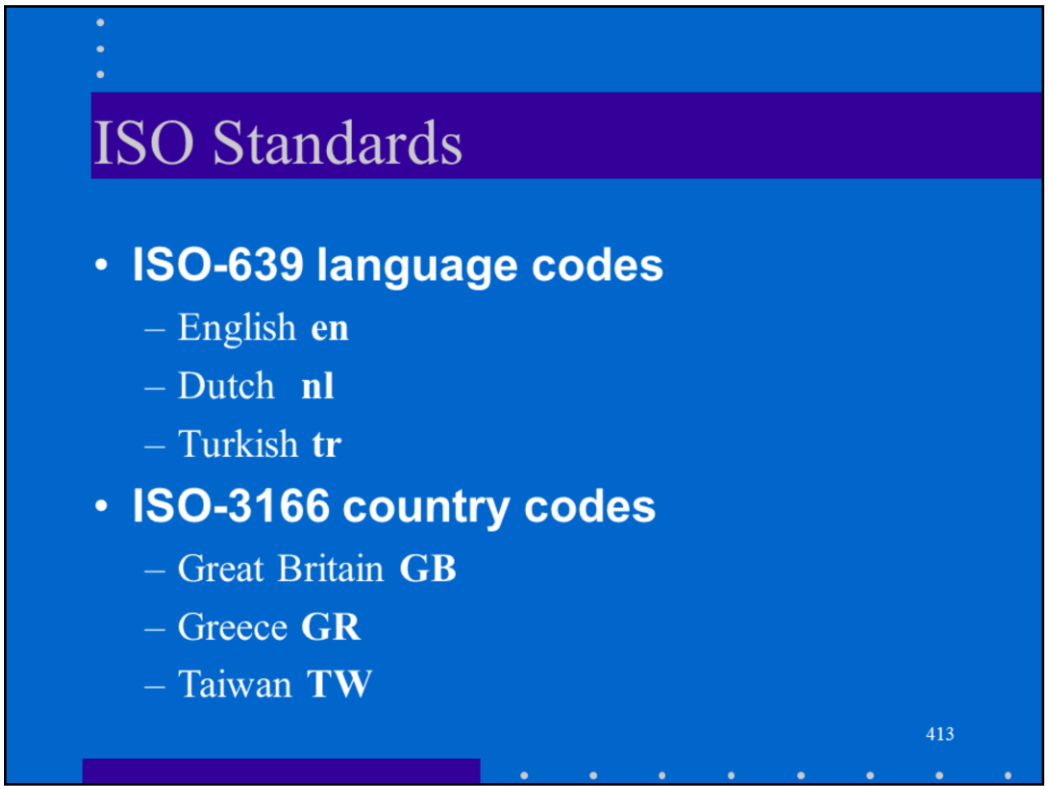
Dates consist of three sets of two digits: day, month and year. Almost every ordering of these appears in one culture or another. The same date has a number of different representations.


Class Locale
• Text formatting in Java can vary from place to place
• Depends on parameter describing location
• Locale object is made up of
– Language
– Location
– Variant
412

Text formatting in Java is implemented in the package java.text. The text formatting in these classes can be made responsive to the location in which the program is executing.

This is achieved by passing a parameter which describes the current location to the methods.

The parameter passed will be an object of class `Locale`. In Java a locale is defined by three variables. The language in use in the locale, the country and any local variant. Variant allows for specialised local dialects and customs. In Great Britain where some people use the language English we might wish to distinguish between a Lancashire variant and a Yorkshire variant.



One daunting aspect of this approach is to determine which countries and languages should be represented. Fortunately there are international standards which deal with this. ISO (International Standards Organisation)-639 assigns a two letter code to languages. ISO-3166 assigns two letter codes to countries. Java makes use of these codes to describe locales.

A list of language codes can be found at

http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt

A list of country codes can be found at

http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

German Currency

```
import java.text.*;
import java.util.*;
class LocaleTest{
    public static void main(String args[])
    {
        Locale loc = new Locale("de", "DE");
        NumberFormat fmt =
                NumberFormat.getCurrencyInstance(loc);
        double wages = 1234.56;
        System.out.println(fmt.format(wages));
    }
}
```

1.234,56 DM

414

This program demonstrates how country and language codes can be used to format text.

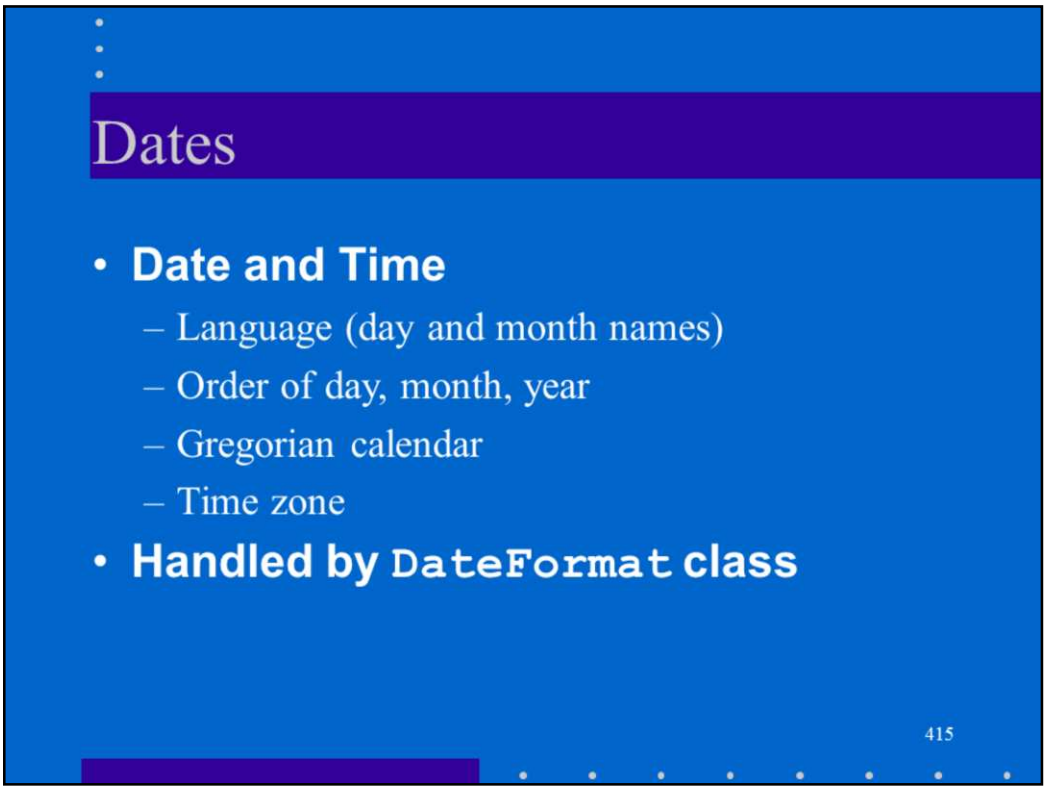
It imports package java.text.

An object of class Locale is created using the language code "de" (German) and the country code "DE" (Germany).

The Locale is used to create an object `fmt` of class `NumberFormat` which is set for currency formatting.

The format is used to print out a double value.

The internal format of this double value is 1234.56. It is output in German format which is 1.234,56 DM. The DM stands for Deutsch Mark which is the German currency unit.



Just as currency varies from country to country, so does the way that a date is represented. There are a number of ways in which the representation of dates varies.

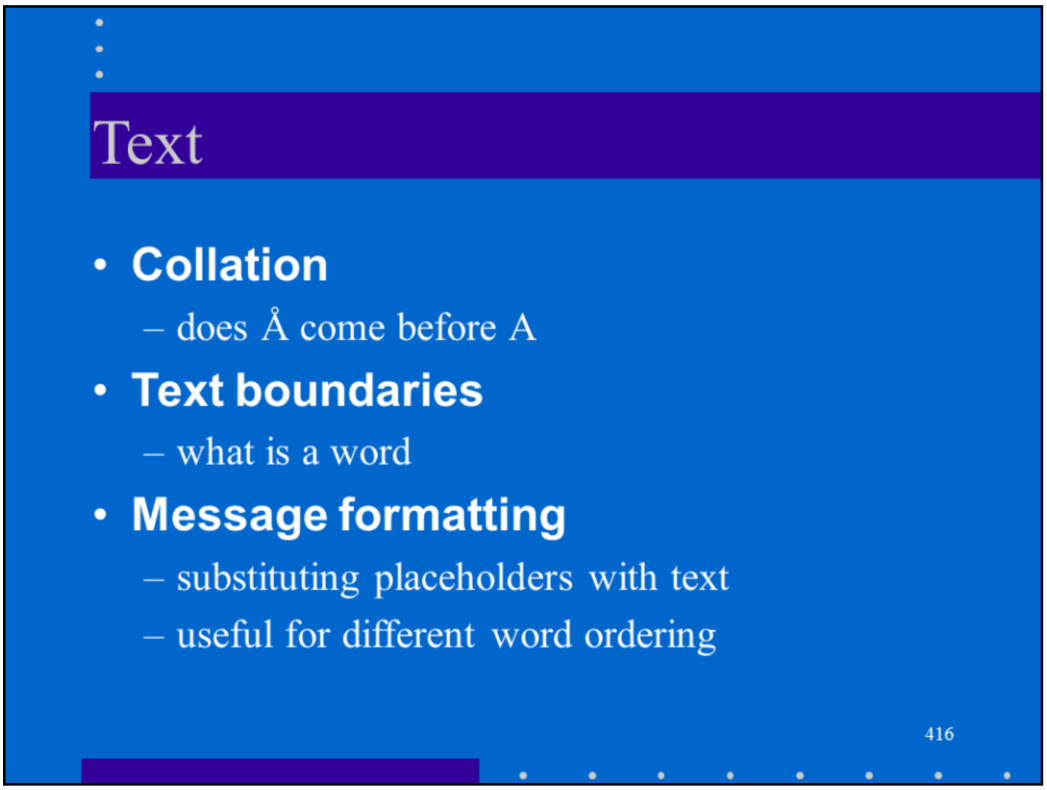
Often a date has its month name written as a word e.g. 25 July 1999. In a different locale the local translation of the month should be used e.g. 25 Juli 1999 in Germany.

As we have seen the day/month/year representation of a date varies from country to country.

In the United Kingdom we use a calendar called the Gregorian (named after Pope Gregory XIII) calendar. We adopted this calendar in 1752 but the rest of Europe had been using it since 1582. Certain groups of people use entirely different calendars which mean that their year number is completely different from the one normally used in the U.K. True internationalisation must allow for this.

Around the world the same moment is recorded as a different time in different time zones. The same event will be assigned different times by people who are geographically separated.

Java handles all these variations in the `DateFormat` class.

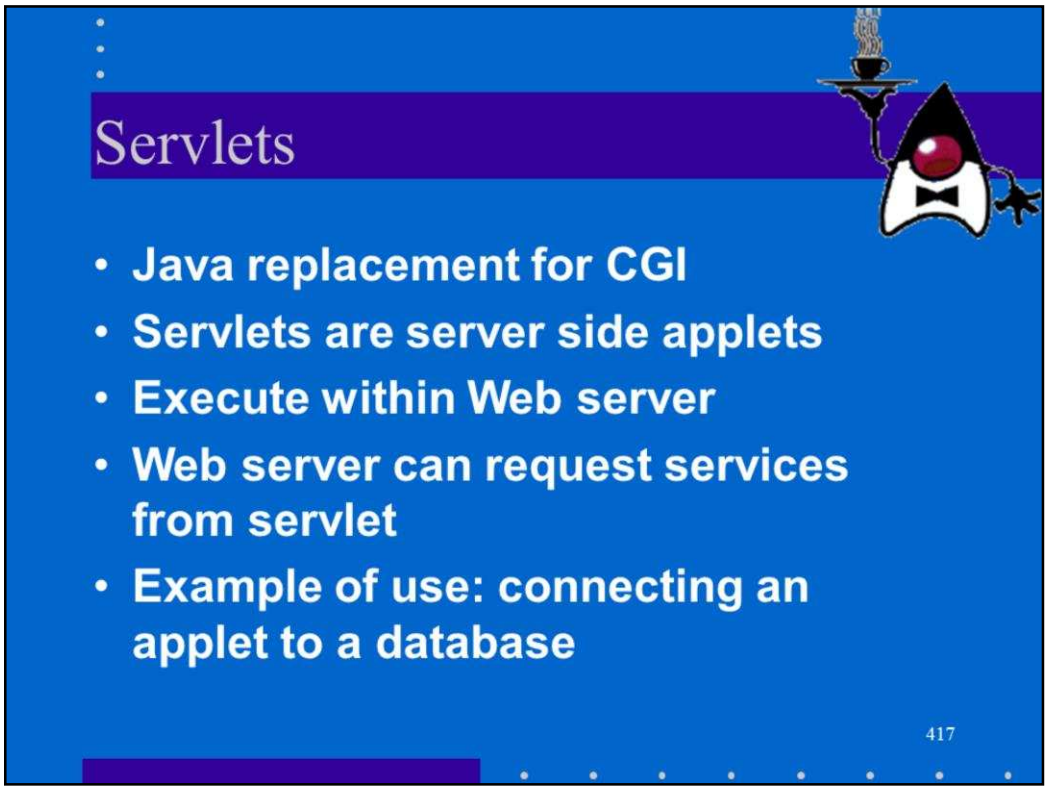


Text poses additional problems. Its all very well having Unicode expand character sets but how does one deal with sorting these extra characters? For example there is a Danish Character Å which looks like an A. If we are sorting should words beginning with Å come before or after words beginning with A? It turns out that Å comes after Z in the collation sequence!

Word processors such as Microsoft Word rely on the fact that they can determine word boundaries. How easy is this in a language which reads right to left?

Java provides mechanisms for dealing with these variations.

Java also copes with situations where word ordering changes from language to language. For example, translating directly from English to German rarely works. To make a German sentence from an English one you often need to move the verb to the end of the sentence. Java provides a macro like facility to achieve this type of translation. These techniques must be used in conjunction with other methods, however, to make a completely accurate translation.

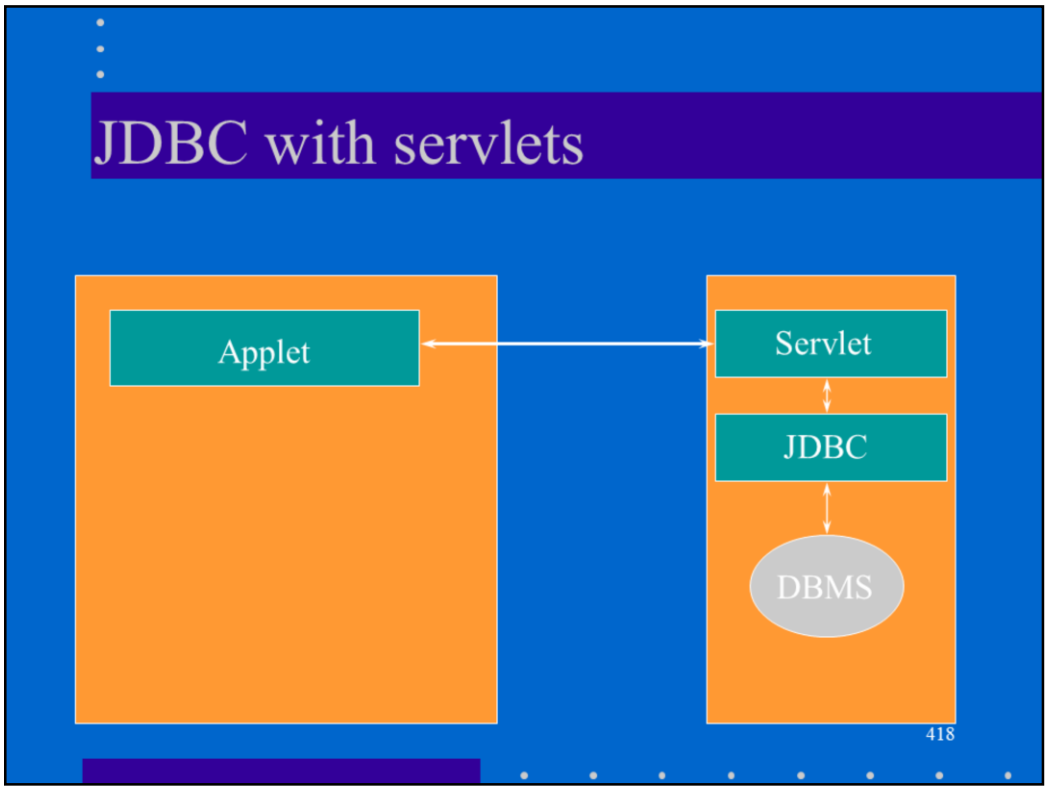

Servlets
• Java replacement for CGI
• Servlets are server side applets
• Execute within Web server
• Web server can request services from servlet
• Example of use: connecting an applet to a database
417

In the early days of the Web the only way one could invoke a program on the server was through the Common Gateway Interface (CGI). Web pages contained a reference to a script written in a language such as Perl or C and a request to run the program was issued when the user clicked on a link or button on the web page. The server then initiated a separate process in which the requested program was executed. This was slow because of the necessity to create a new process, an operation which is always resource heavy for an operating system. Servlets do not run in their own process space and are therefore less of a drain on the server host's operating system.

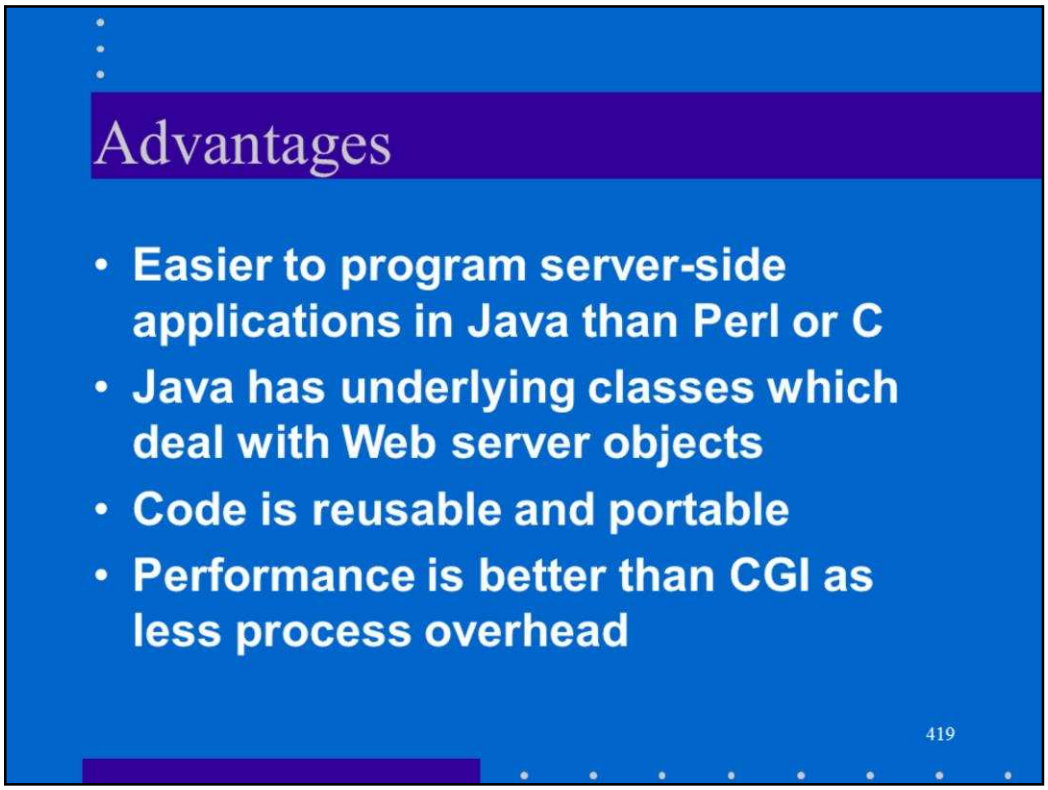
Servlets are an alternative to CGI scripts. They are like applets which run on the server rather than the client. Due to the way they are implemented they are much more efficient than CGI scripts with the advantage that they are portable.

"Organic Online, a Web development company in San Francisco conducted internal tests that indicate significantly improved performance with both Fast CGI and Java Servlets on JavaWeb Server or Jigsaw (the Java-based server from the World Wide Web Consortium), over CGI-bin programs. Testing was conducted with a Sparc 5 with 64 MB RAM and a 10BaseT network connection. Java Servlets and Fast CGI processed about 3-4 requests per second, compared to 1/4 request per second with regular CGI."

Details of servlets can be found at http://java.sun.com/products/java-server/servlets/index.html.



Earlier in the lecture the slides on JDBC highlighted problems that might occur if one tried to put JDBC access into an applet. These can be overcome using servlet technology. The applet, without any JDBC classes or need to store data on the client, communicates with a servlet which can use JDBC to communicate with a database. Alternatively, the servlet can communicate with remote databases on other machines.


Advantages
• Easier to program server-side applications in Java than Perl or C
• Java has underlying classes which deal with Web server objects
• Code is reusable and portable
• Performance is better than CGI as less process overhead
419

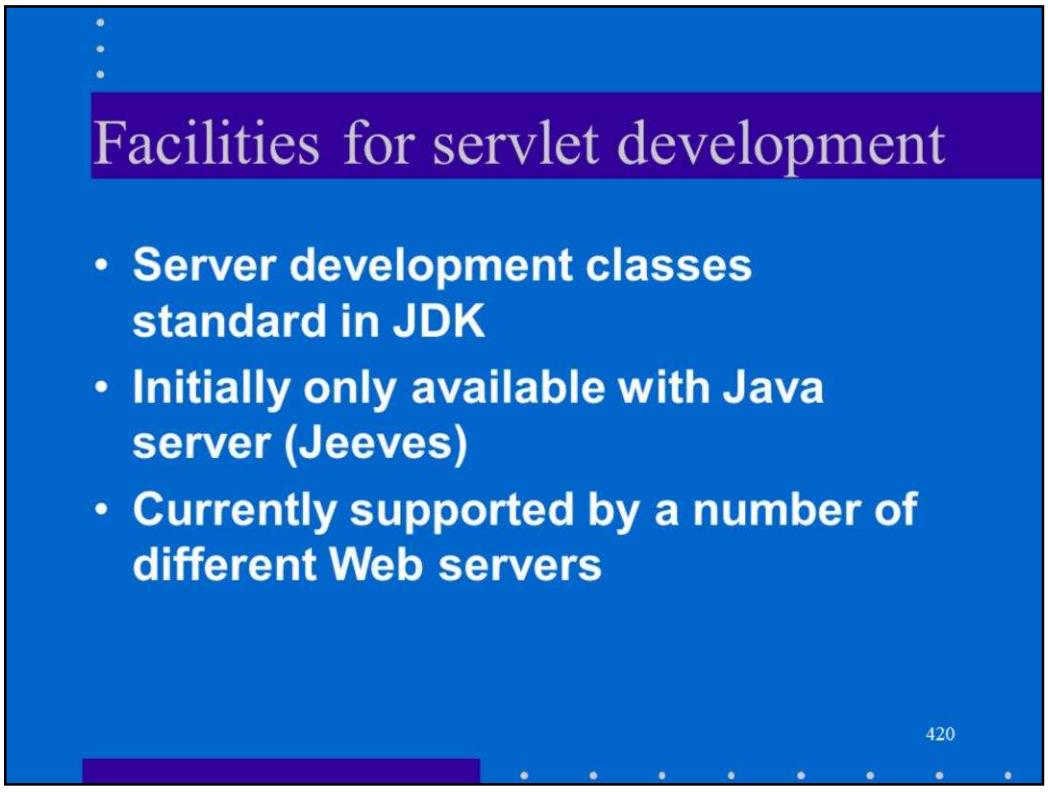
There are a number of arguments supporting the use of servlets.

If you accept the argument that it is easier to program in Java than C or Perl then there is an immediate gain to using servlets in preference to CGI scripts, simply because the effort required to produce them is less.

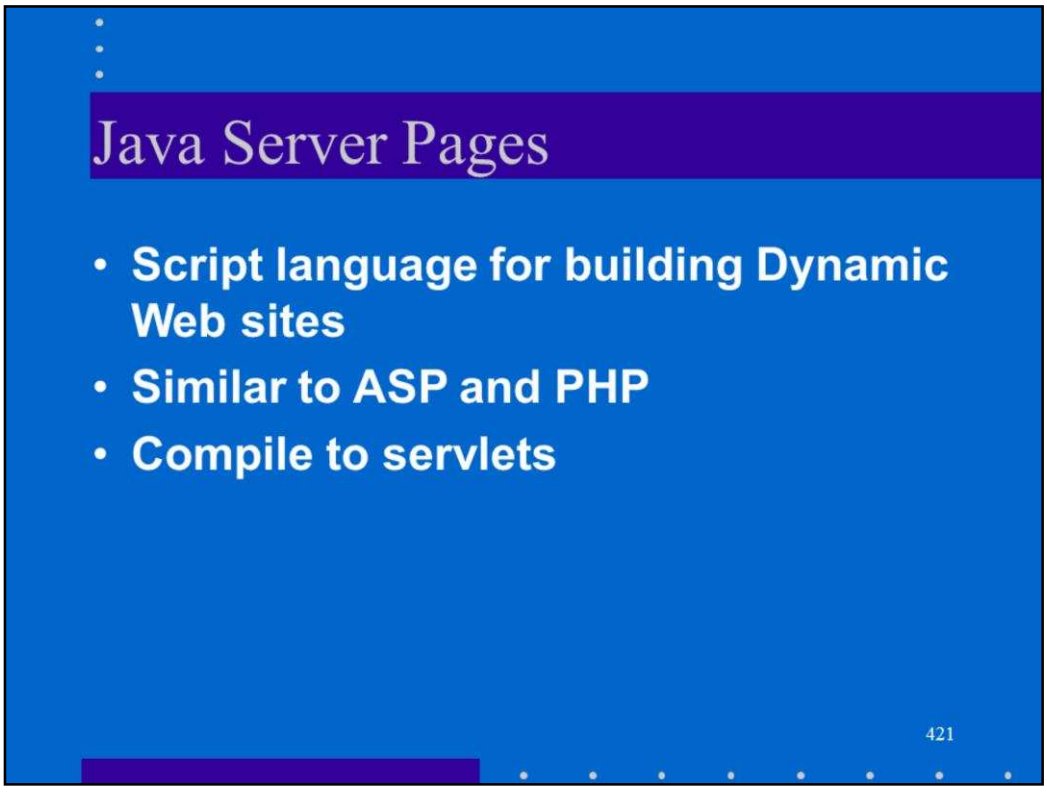
The effort is reduced even further because there are Java packages which make it easier to communicate with the Web server than standard C or Perl procedure calls.

As always with Java, the code is potentially portable.

Servlets run in the same address space as the server and so require less operating system overhead than conventional CGI scripts.

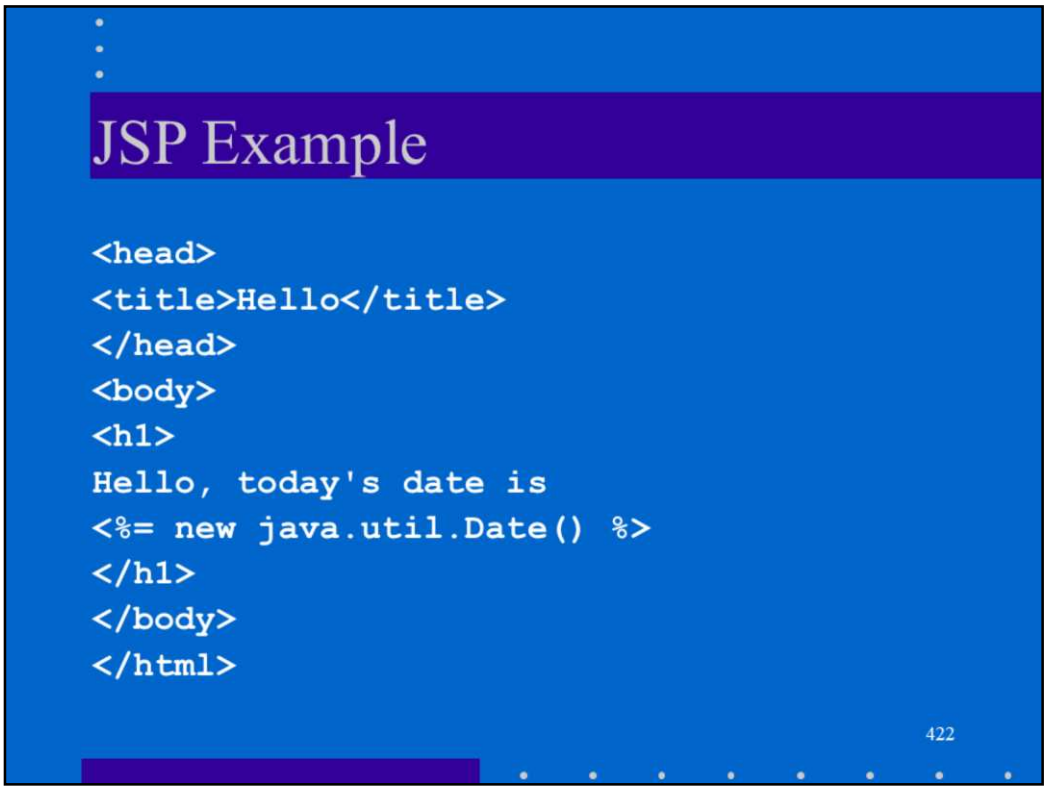

Facilities for servlet development
• Server development classes standard in JDK
• Initially only available with Java server (Jeeves)
• Currently supported by a number of different Web servers
420

Initially servlets could only be implemented in conjunction with the Java Web server. Now a long list of Web servers support the implementation of servlets. Visit http://java.sun.com/products/servlet/industry.html#servers to see a list of Web servers which support servelts

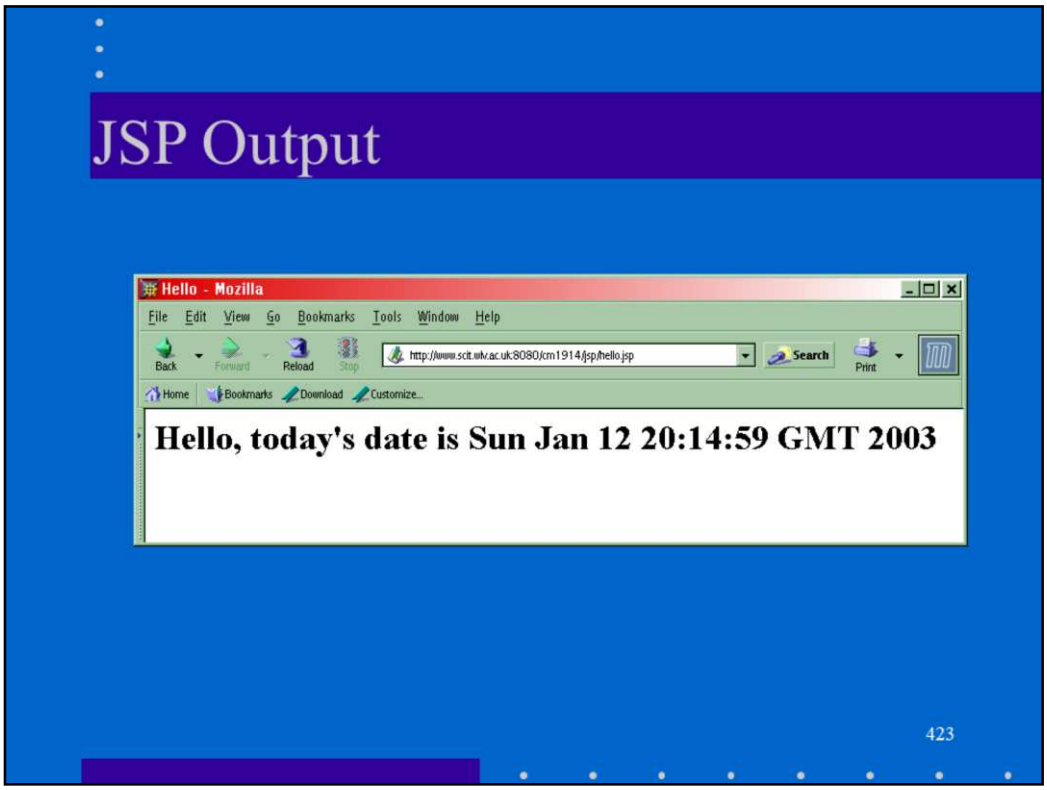


http://java.sun.com/products/jsp/

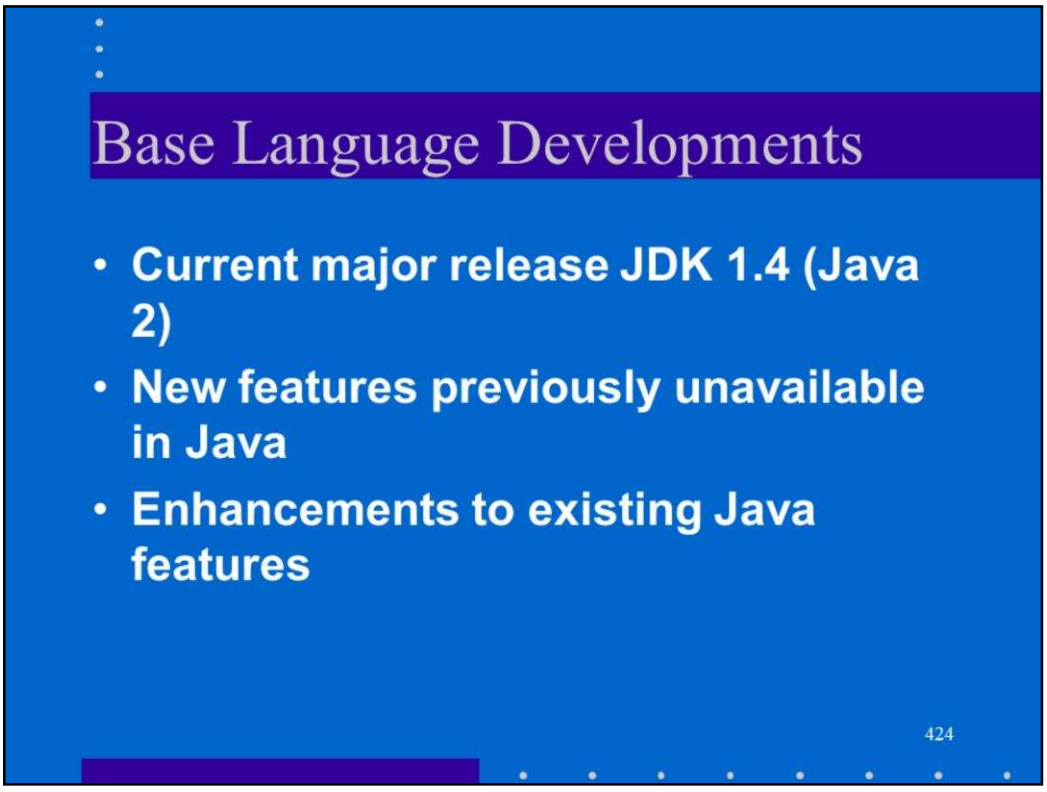
JavaServer Pages technology is an extension of the Java™ Servlet technology. Servlets are platform-independent, 100% pure Java server-side modules that fit seamlessly into a Web server framework and can be used to extend the capabilities of a Web server with minimal overhead, maintenance, and support. Unlike other scripting languages, servlets involve no platform-specific consideration or modifications; they are Java application components that are downloaded, on demand, to the part of the system that needs them. Together, JSP technology and servlets provide an attractive alternative to other types of dynamic Web scripting/programming that offers platform independence, enhanced performance, separation of logic from display, ease of administration, extensibility into the enterprise and most importantly, ease of use.



This text is placed in a file called hello.jsp. When the JSP enabled Web server is asked to deliver the file it executes the Java in the <% %> tags. This is a call to a standard Java function which returns the date. The = instructs JSP to insert the text of this date into the HTML.



This demonstrates the effect of accessing the JSP page


Base Language Developments
• Current major release JDK 1.4 (Java 2)
• New features previously unavailable in Java
• Enhancements to existing Java features
424

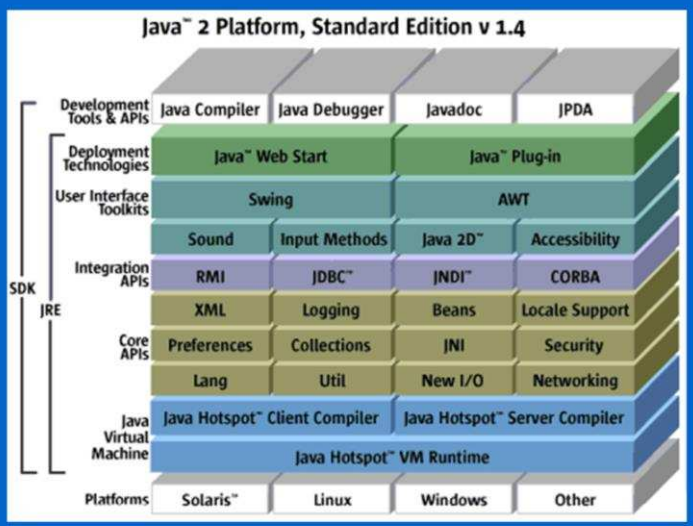
JDK 1.1 contained major improvements and enhancements of JDK 1.0. All the subsequent releases of this JDK (JDK 1.1.1 through JDK 1.1.7) represented bug fixes of the basic JDK 1.1.

When JDK 1.2 was released the changes were so significant that the language it supported was referred to as Java 2.

The current release of the JDK is JDK 1.4 (Java 2). Much of the material in this lecture has covered Java developments which have been made available following the release of JDK 1.1. Most of these developments are standard content in JDK 1.2 and subsequent releases.

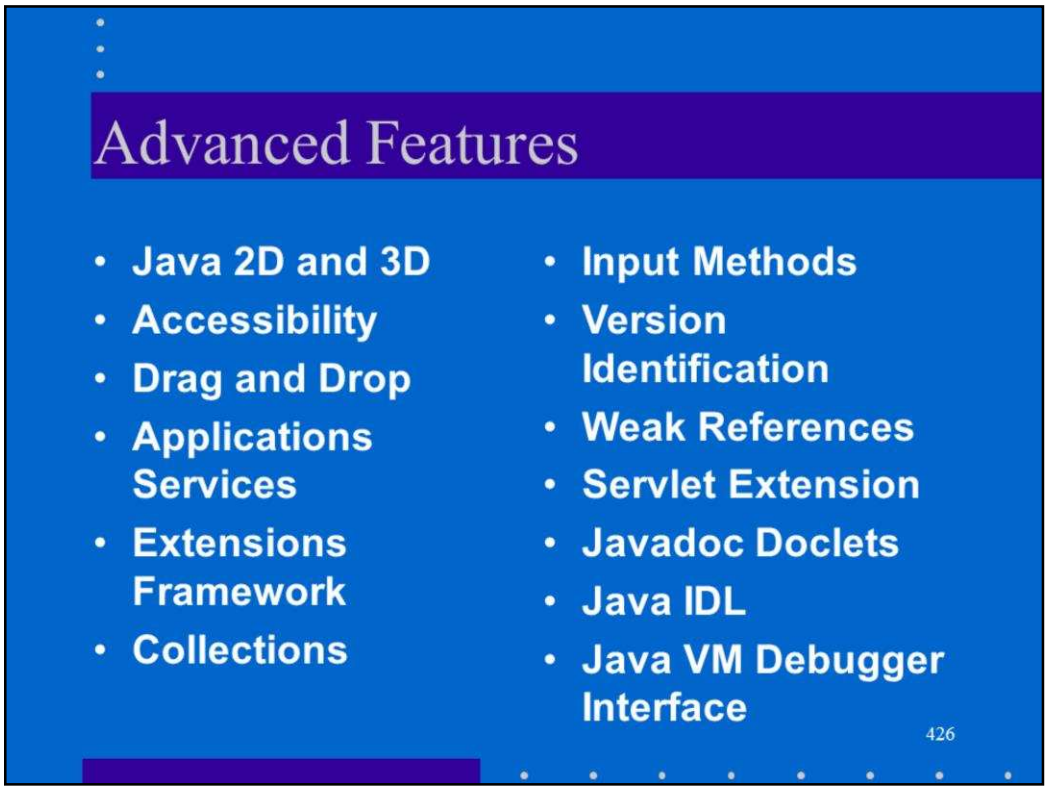
A full release of Java 2 (still 2 despite new JDK releases) is available. As usual it is accessible by free download across the Internet.

Java 2 Platform Standard Ed 1.4



425

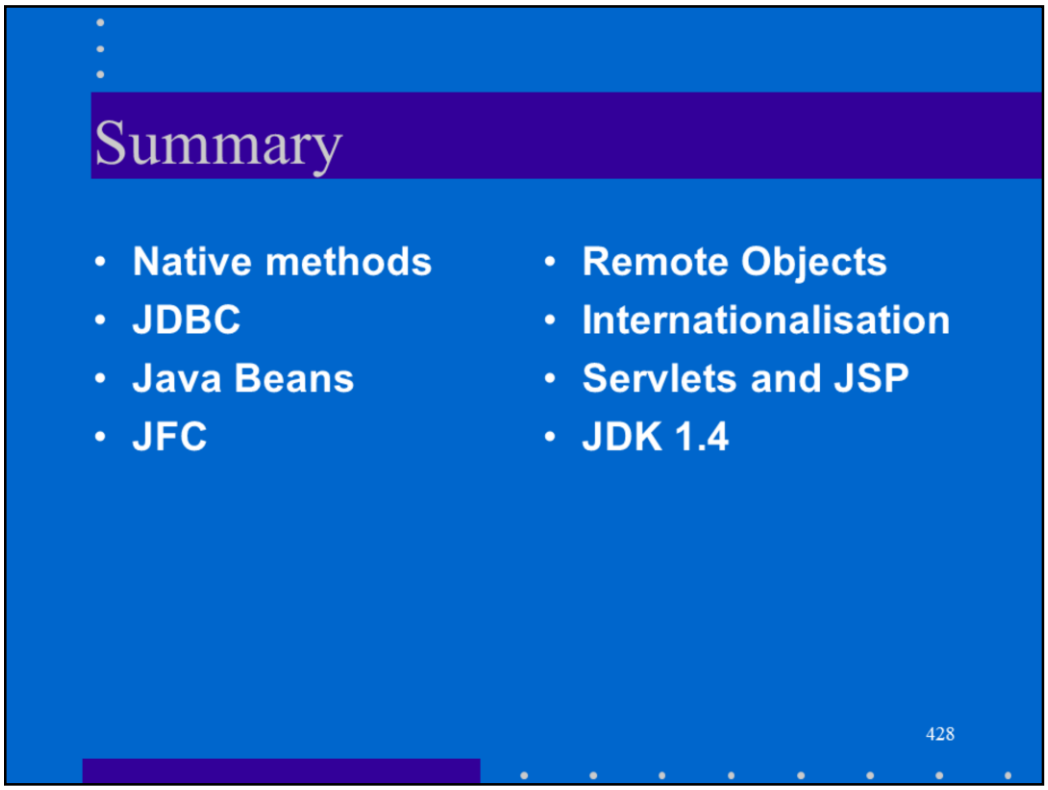
From http://java.sun.com/j2se/1.4/



Full details and download of the current release of JDK 1.3 can be found at: http://java.sun.com/j2se/1.4/

Enhancements

- Java Beans
- RMI
- Object Serialisation
- Java Sound
- Java Archive
- Java Native Interface
- Memory compression for loaded classes
- Solaris thread support
- Faster memory allocation
- Faster garbage collection



The development of Java continues apace. A year from now Java will have a whole host of new features not covered in these lectures. The history of Java is a short one but it has already shown a capacity for adapting in the light of user criticisms and evolving exciting new packages.